Imperial College London

The Next 700 Domain Specific Languages

Nicolas Wu

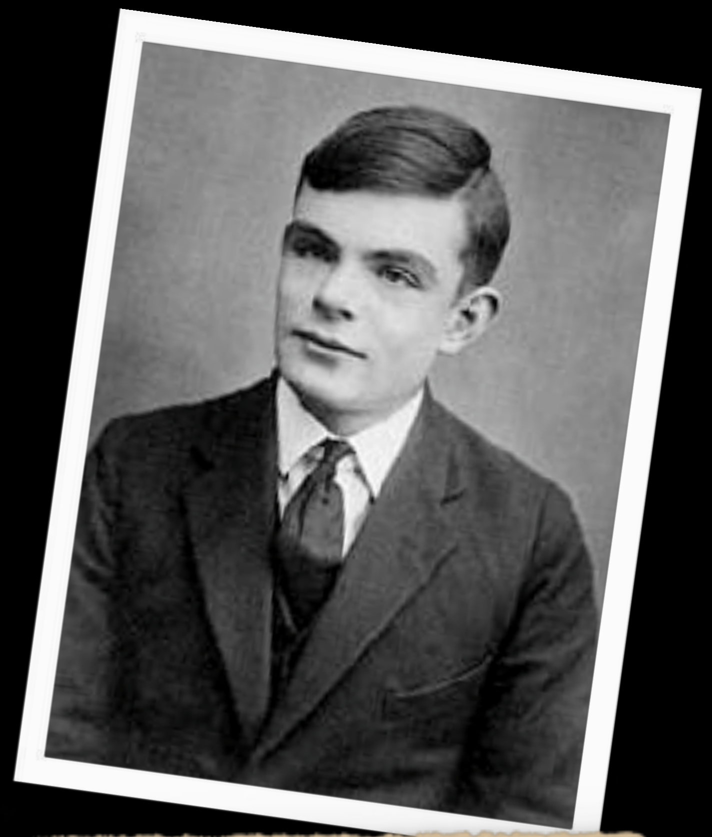
Imperial College London

Peter Landin Seminar, BCS 2022

functional

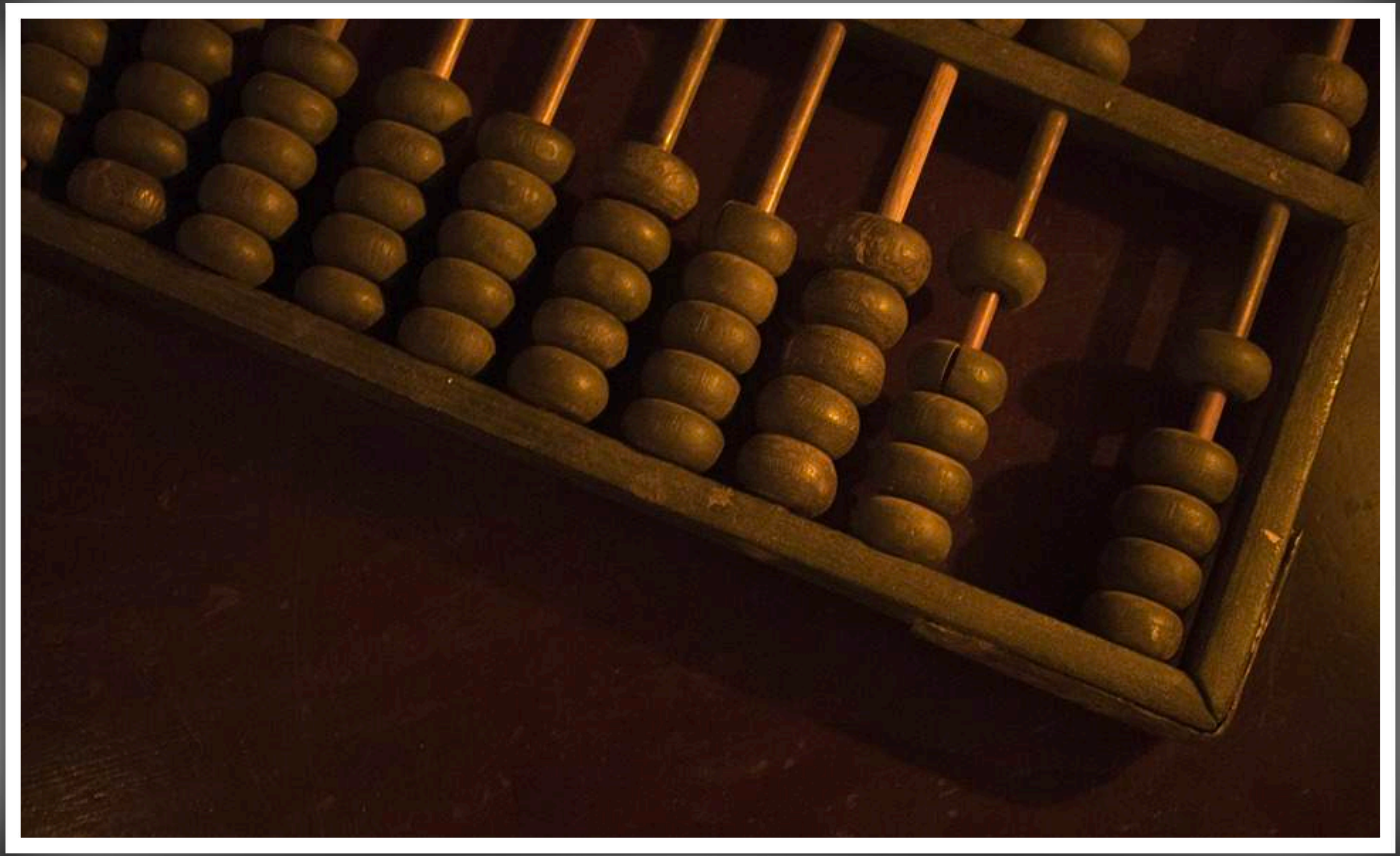
Alonzo Church

`> imperative`



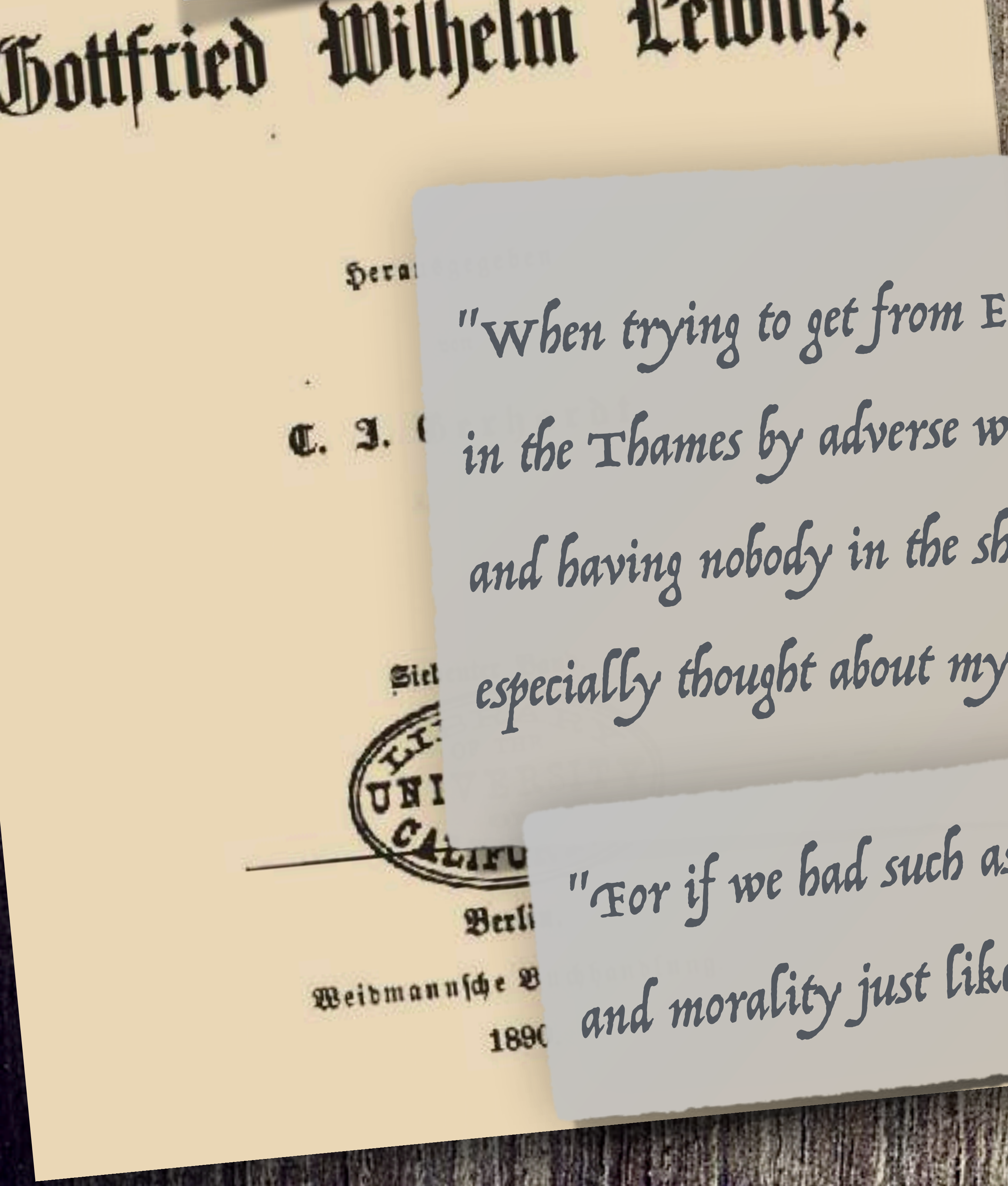
Alan Turing

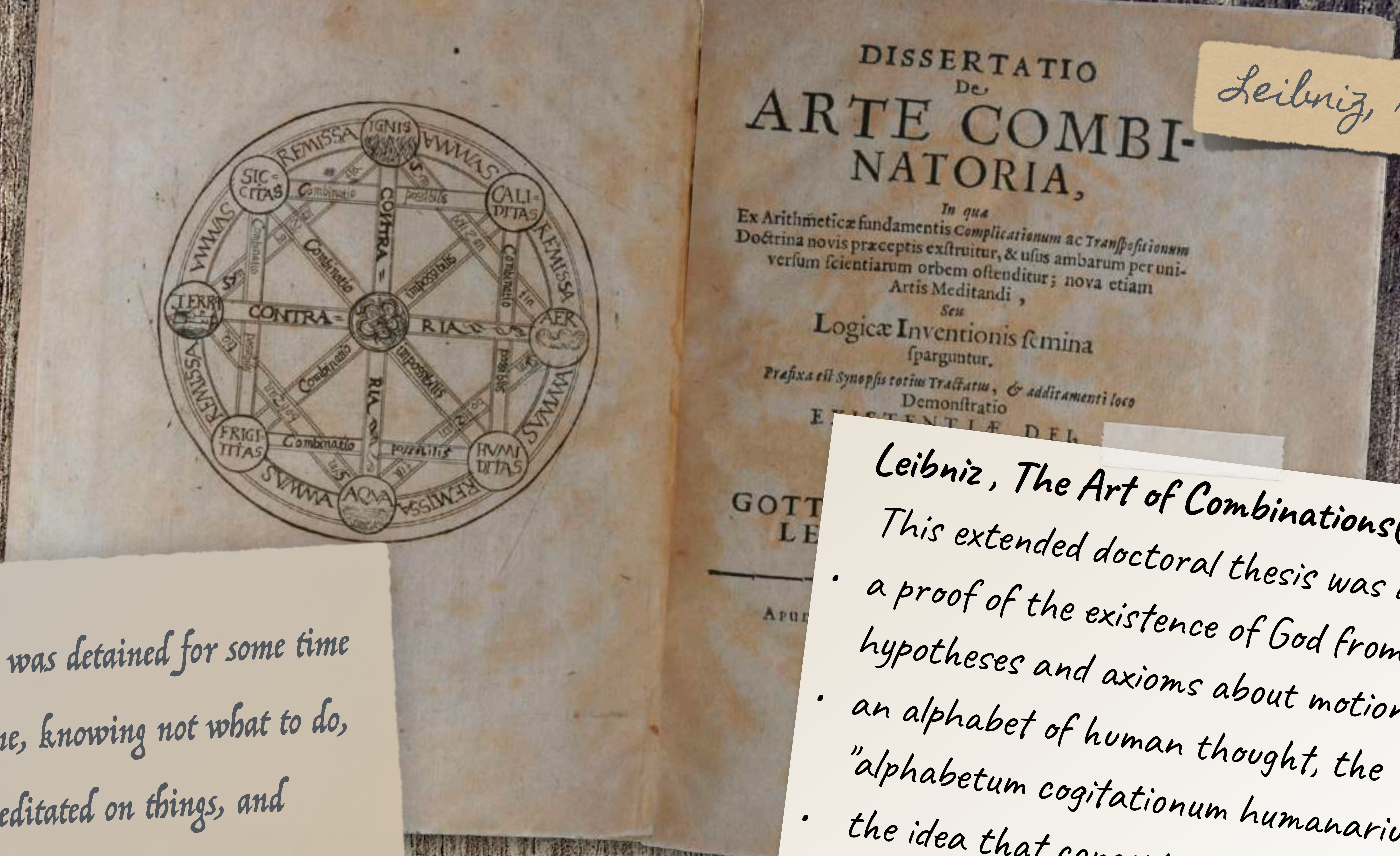
Before Computers




London, 1666

Gottfried Leibniz
1646 - 1716


phil
Gottfried Wilhelm Leibniz.
Hera
C. J.
Sie
Berli
Weidmannsche B
1890


DISSERTATIO
DE
ARTE COMBINATORIA,
In qua
Ex Arithmeticæ fundamentis Complicationum ac Transpositionum Doctrina novis præceptis exstruitur, & usus ambarum per universum scientiarum orbem ostenditur; nova etiam Artis Meditandi, Seu Logicæ Inventionis semina sparguntur.
Præfixa est Synopsis totius Tractatus, & additamenti loco Demonstratio
EXISTENTIÆ DEI
GOTT
LE
APUD

Leibniz, 1666

Leibniz, The Art of Combinations (1666)

This extended doctoral thesis was about:

- a proof of the existence of God from some hypotheses and axioms about motion
- an alphabet of human thought, the "alphabetum cogitationum humanarium"
- the idea that concepts are combinations of a core set of concepts

"When trying to get from England to Holland, I was detained for some time in the Thames by adverse winds. During that time, knowing not what to do, and having nobody in the ship except sailors, I meditated on things, and especially thought about my old design of a rational language [...]"

"For if we had such as I imagine, we could reason about metaphysics and morality just like we do about geometry and analysis [...]"

letter from Leibniz to Jean Galois, 1677

"For if praise is given to the men who have determined the number of regular solids [...] how much better will it be to bring under mathematical laws human reasoning, which is the most excellent and useful thing we have." (Davis, 2018)

Emil du Bois-Reymond
1818 - 1896

"Ignoramus et ignorabimus"
We do not know and we shall not know

du Bois-Reynold, 1872,
Congress of German Scientists and Physicians

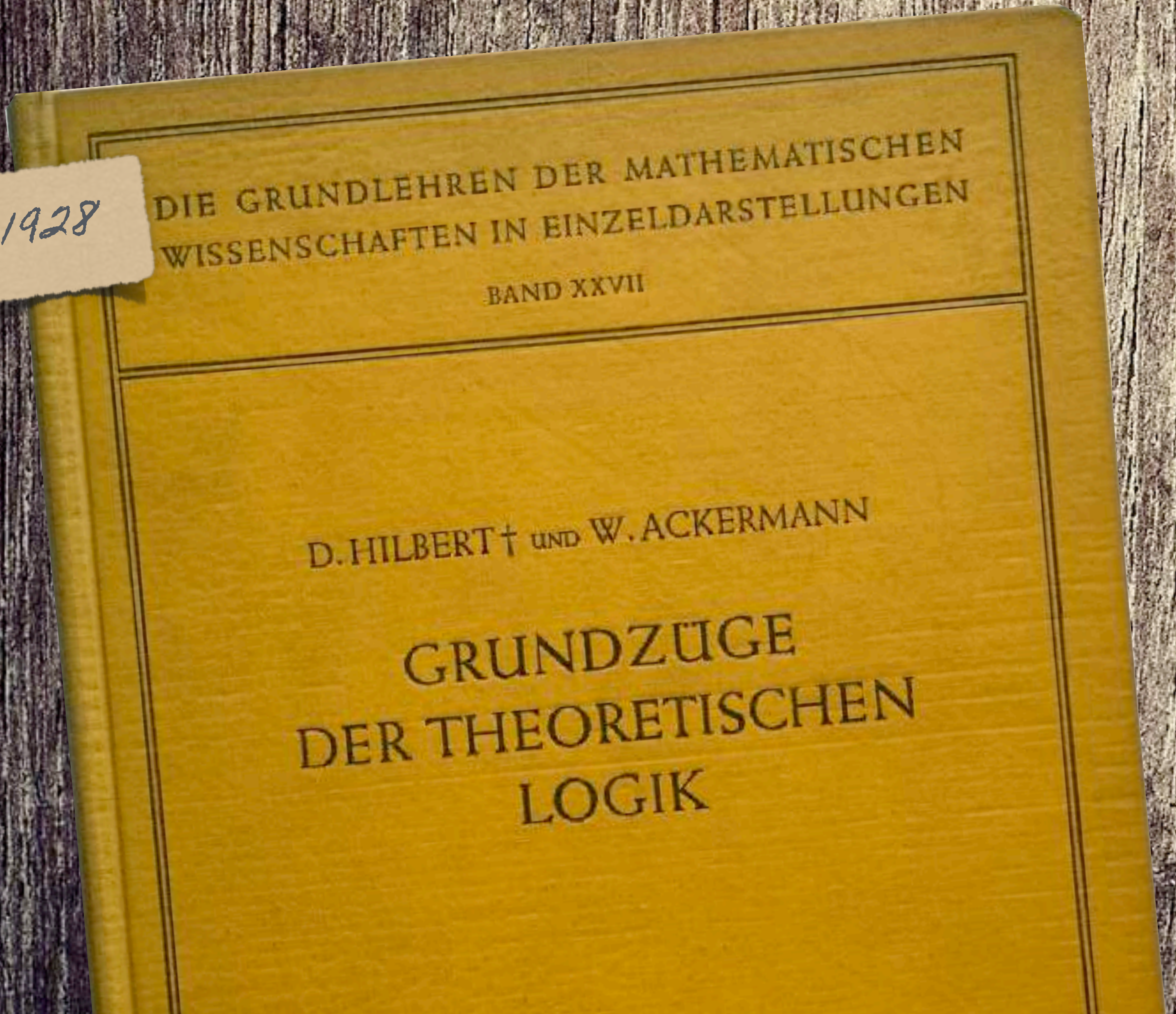
"Wir müssen wissen. Wir werden wissen!"
We must know. We shall know!

Hilbert, 1930,
Congress of German Scientists and Physicians

David Hilbert
1862 - 1943

Wilhelm Ackermann
1896 - 1962

1928



Hilbert and Ackerman (1928)

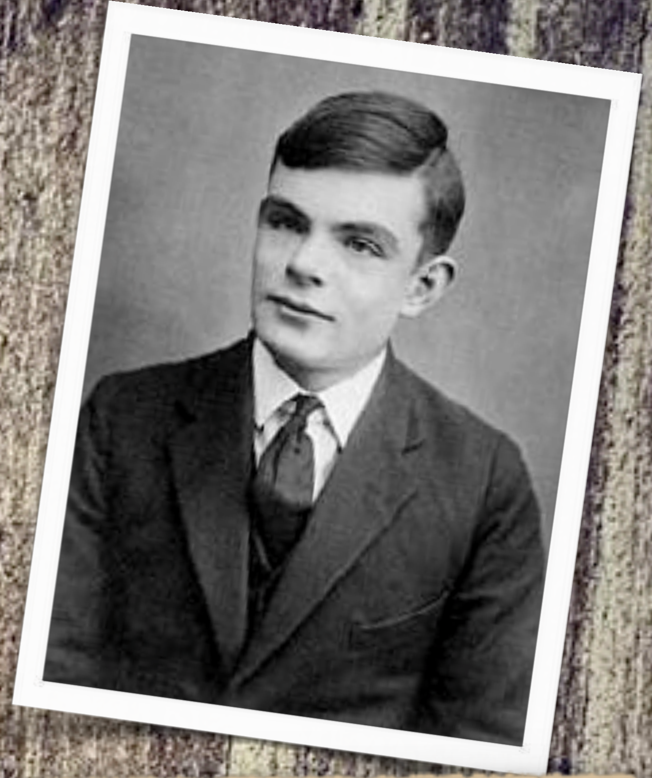
Formalised the Entscheidungsproblem,
the "decision problem":
Given an input statement in first order predicate calculus, is there an algorithm that answers either "yes" or "no" if the statement is provable?

Kurt Gödel
1906 - 1978

Gödel (1931)
showed that any reasonable system of logic has true statements that cannot be proven within that system

but if the Entscheidungsproblem holds true, then perhaps an algorithm can prove statements?

Alonzo Church



Alan Turing

Church (1936) and Turing (1936)
Independently answered the Entscheidungsproblem:
No! Such an algorithm is not possible

1931

Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I[1]).

Von Kurt Gödel in Wien.

1.

Die Entwicklung der Mathematik in der Richtung zu größerer Exaktheit hat bekanntlich dazu geführt, daß weite Gebiete von ihr formalisiert wurden, in der Art, daß das Beweisen nach einigen wenigen mechanischen Regeln vollzogen werden kann. Die umfassendsten derzeit aufgestellten formalen Systeme sind das System der Principia Mathematica (PM)[2]) einerseits, das Zermelo-Fraenkelsche (von J. v. Neumann weiter ausgebildete) Axiomensystem der Mengenlehre[3]) andererseits. Diese beiden Systeme sind so weit, daß alle heute in der Mathematik angewendeten Beweismethoden in ihnen formalisiert, d. h. auf einige wenige Axiome und Schlußregeln zurückgeführt sind. Es liegt daher die Vermutung nahe, daß diese Axiome und Schlußregeln dazu ausreichen, alle mathematischen Fragen, die sich in den betreffenden Systemen überhaupt formal ausdrücken lassen, auch zu entscheiden. Im folgenden wird gezeigt, daß dies nicht der Fall ist, sondern daß es in den beiden angeführten Systemen sogar relativ einfache Probleme aus der Theorie der gewöhnlichen ganzen Zahlen gibt[4]), die sich aus den Axiomen nicht

[1]) Vgl. die im Anzeiger der Akad. d. Wiss. in Wien (math.-naturw. Kl.) 1930, Nr. 19 erschienene Zusammenfassung der Resultate dieser Arbeit.

[2]) A. Whitehead und B. Russell, Principia Mathematica, 2. Aufl., Cambridge 1925. Zu den Axiomen des Systems PM rechnen wir insbesondere auch: Das Unendlichkeitsaxiom (in der Form: ...

1936

The Journal of Symbolic Logic
Volume I, Number 1, March 1936

A NOTE ON THE ENTSCHEIDUNGSPROBLEM

ALONZO CHURCH

In a recent paper[1] the author has proposed a definition of the commonly used term "effectively calculable" and has shown on the basis of this definition that the general case of the Entscheidungsproblem is unsolvable in any system of symbolic logic which is adequate to a certain portion of arithmetic and is ω-consistent. The purpose of the present note is to outline an extension of this result to the engere Funktionenkalkül of Hilbert and Ackermann.[2]

In the author's cited paper it is pointed out that there can be associated recursively with every well-formed formula[3] a recursive enumeration of the formulas into which it is convertible.[3] This means the existence of a recursively defined function a of two positive integers such that, if y is the Gödel representation of a well-formed formula Y then $a(x, y)$ is the Gödel representation of the xth formula in the enumeration of the formulas into which Y is convertible.

Consider the system L of symbolic logic which arises from the engere Funktionenkalkül by adding to it: as additional undefined symbols, a symbol 1 for the number 1 (regarded as an individual), a symbol = for the propositional function = (equality of individuals), a symbol s for the arithmetic function $x+1$, a symbol a for the arithmetic function a described in the preceding paragraph, and symbols $b_1, b_2, \cdots, b_k$ for the auxiliary arithmetic functions which are employed in the recursive definition of ...

1936

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable ... written down by a machine.

Alonzo Church

The definition of "effectively calculable"

A proof that converting a term A into B is undecidable

The Lambda Calculus

λ-calculus is about bindings and substitutions

Instead of

$f(x) = 5x + 3$

We can isolate f:

$f = \lambda x . 5x + 3$

Church numerals:

$1 = \lambda s z . s z$

$2 = \lambda s z . s (s z)$

$3 = \lambda s z . s (s (s z))$

...

$m + n = \lambda s z . m s (n s z)$

$\_+\_ = \lambda m n s z . m s (n s z)$

1932

A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

BY ALONZO CHURCH.[2]

...roduction. In this paper we present a set of postulates for the ... of formal logic, in which we avoid use of the free, or real, ...nd in which we introduce a certain restriction on the law of ...cluded middle as a means of avoiding the paradoxes connected with the mathematics of the transfinite.

Our reason for avoiding use of the free variable is that we require that every combination of symbols belonging to our system, if it represents a proposition at all, shall represent a particular proposition, unambiguously, and without the addition of verbal explanations. That the use of the free variable involves violation of this requirement, we believe is readily seen. For example, the identity

(1) $$a(b+c) = ab+ac$$

in which a, b, and c are used as free variables, does not state a definite proposition unless it is known what values may be taken on by these variables, and this information, if not implied in the context, must be given by a verbal addition. The range ...

1936

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$[2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly the condition that the function f be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

1936

The Journal of Symbolic Logic
Volume 1, Number 1, March 1936

A NOTE ON THE ENTSCHEIDUNGSPROBLEM

ALONZO CHURCH

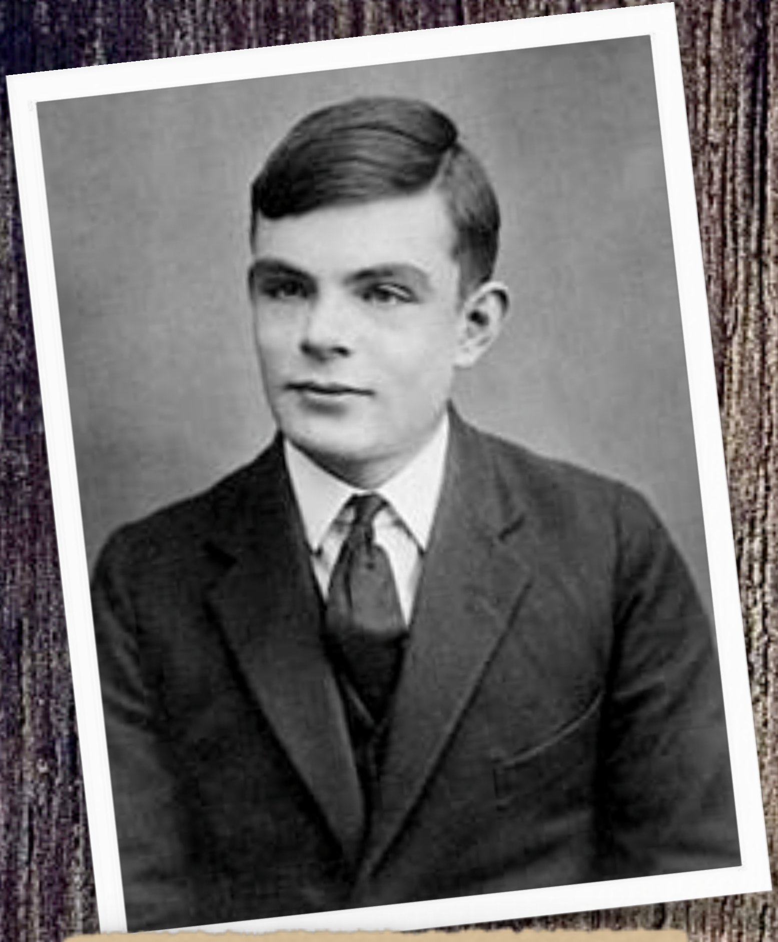
In a recent paper[1] the author has proposed a definition of th... used term "effectively calculable" and has shown on the basis of t... that the general case of the Entscheidungsproblem is unsolvable i... of symbolic logic which is adequate to a certain portion of arith... ω-consistent. The purpose of the present note is to outline an ext... result to the engere Funktionenkalkül of Hilbert and Ackermann.[2]

In the author's cited paper it is pointed out that there can... recursively with every well-formed formula[3] a recursive enumera... mulas into which it is convertible.[3] This means the existence o... defined function a of two positive integers such that, if y is the G... tation of a well-formed formula Y then $a(x, y)$ is the Gödel repre... xth formula in the enumeration of the formulas into which Y is c...

Consider the system L of symbolic logic which arises from th... tionenkalkül by adding to it: as additional undefined symbols... the number 1 (regarded as an individual), a symbol = for th... function = (equality of individuals), a symbol s for the ari... $x+1$, a symbol a for the arithmetic function a described in th... graph, and symbols $b_1, b_2, \cdots, b_k$ for the auxiliary arithmeti... employed in the recursive definition of a; and as additional ...

010100100100011101010101001010101001001011100101001010011001

Turing Machines



Alan Turing

Automatic machines.

If at each stage the motion of a machine (in the sense of § 1) is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a*-machine).

Computing machines.

If an *a*-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine.

We can show further that *there can be no machine* $\mathcal{E}$ *which, when supplied with the S.D of an arbitrary machine* $\mathcal{M}$, *will determine whether* $\mathcal{M}$ *ever prints a given symbol* (0 *say*).

The results of § 8 have some important applications. In particular, they can be used to show that the Hilbert Entscheidungsproblem can have no solution. For the present I shall confine myself to proving this particular

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

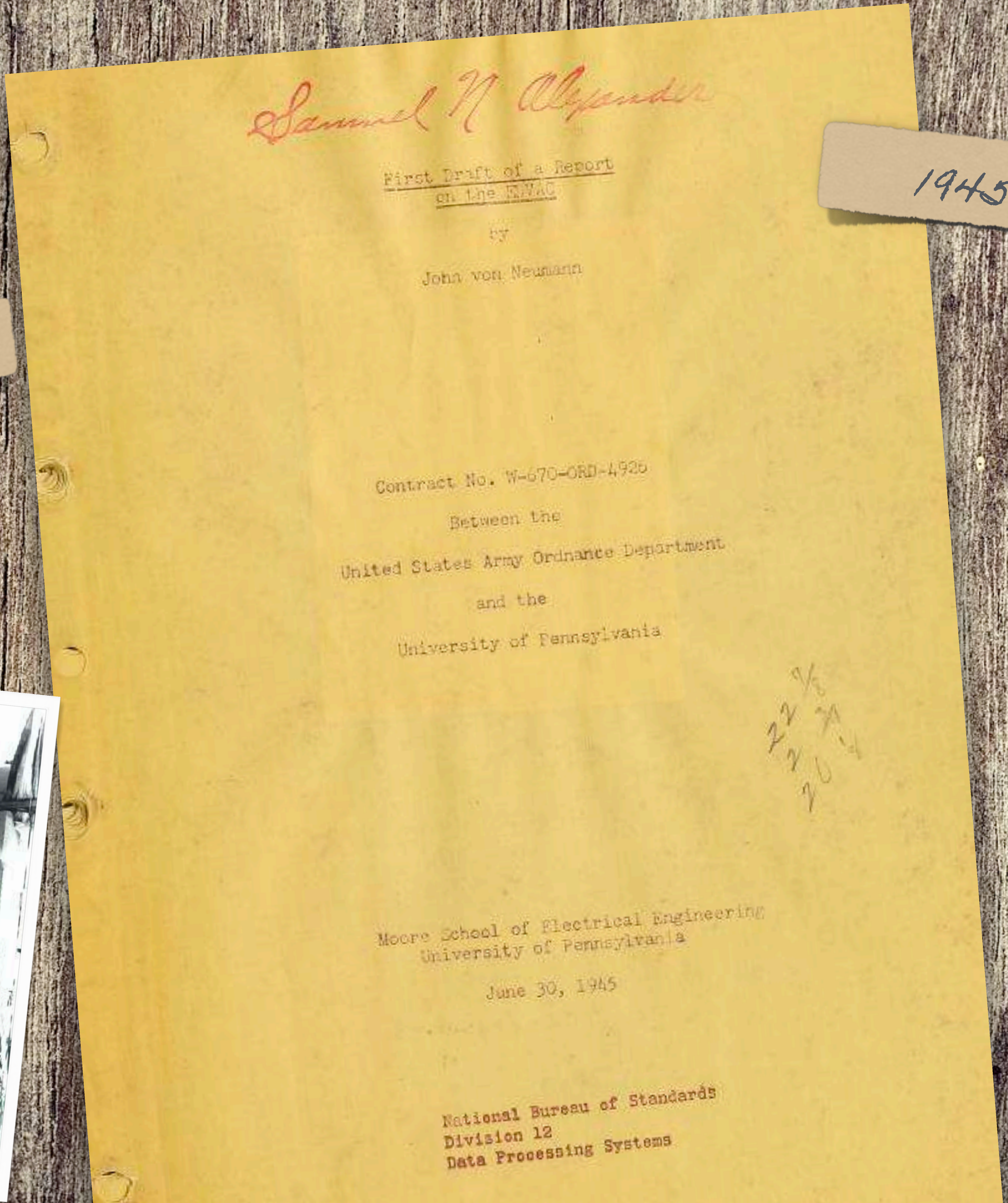
[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a ... of the theory of functions ...

Turing showed construction of a formula that takes in a Turing machine as input and is provable only if the machine halts

but he also showed that no machine can determine if an arbitrary machine halts

John von Neumann
1903 - 1957



Samuel N Alexander

First Draft of a Report
on the EDVAC

by

John von Neumann

Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania

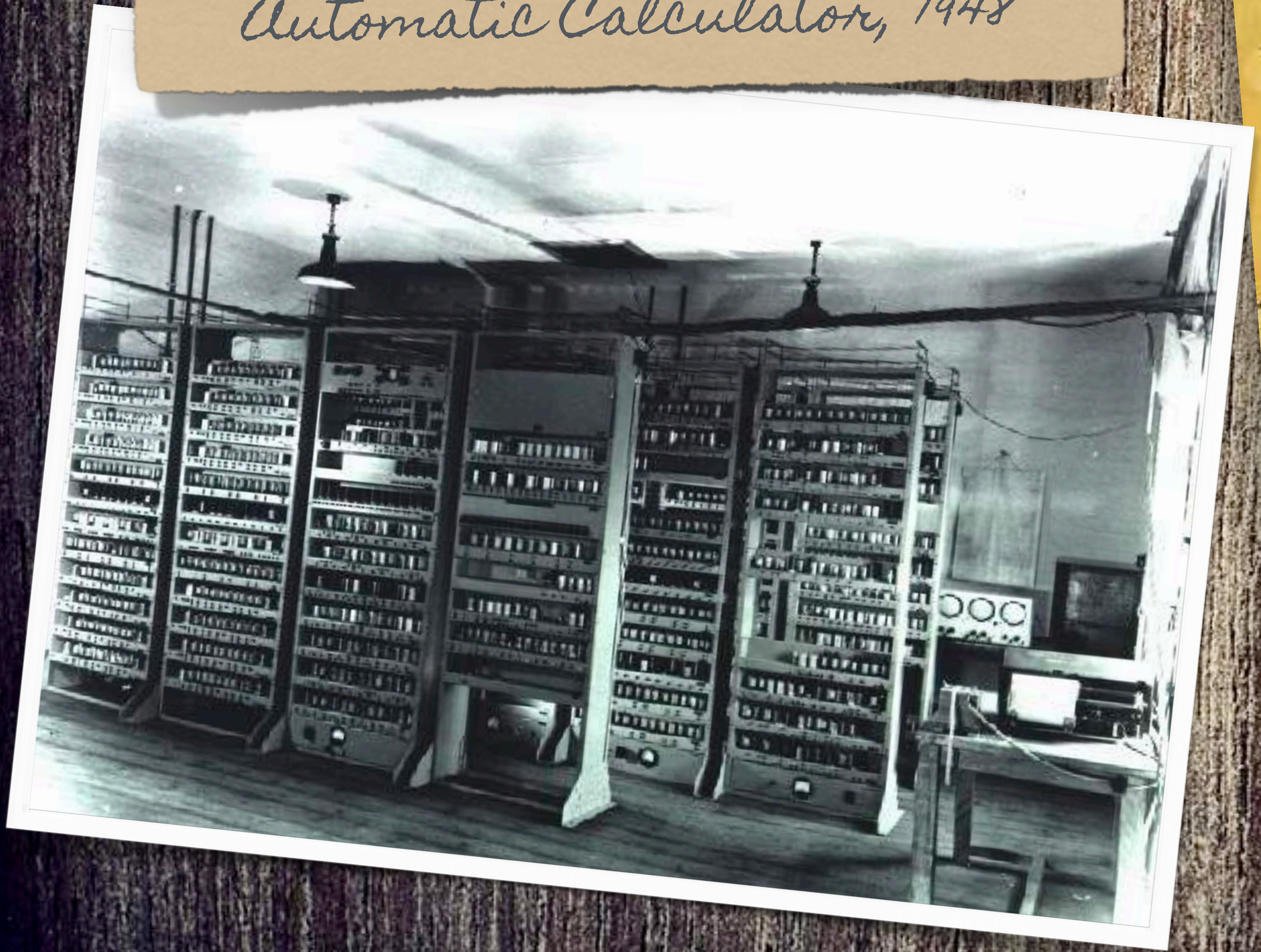
Moore School of Electrical Engineering
University of Pennsylvania

June 30, 1945

National Bureau of Standards
Division 12
Data Processing Systems

1945

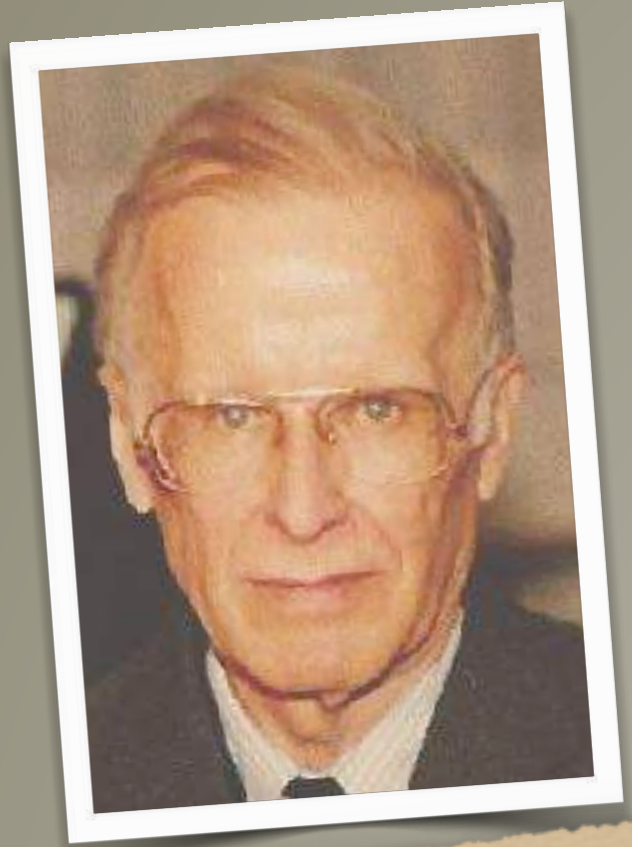
Electronic Delay Storage
Automatic Calculator, 1948



Electronic Discrete Variable
Automatic Computer, 1949

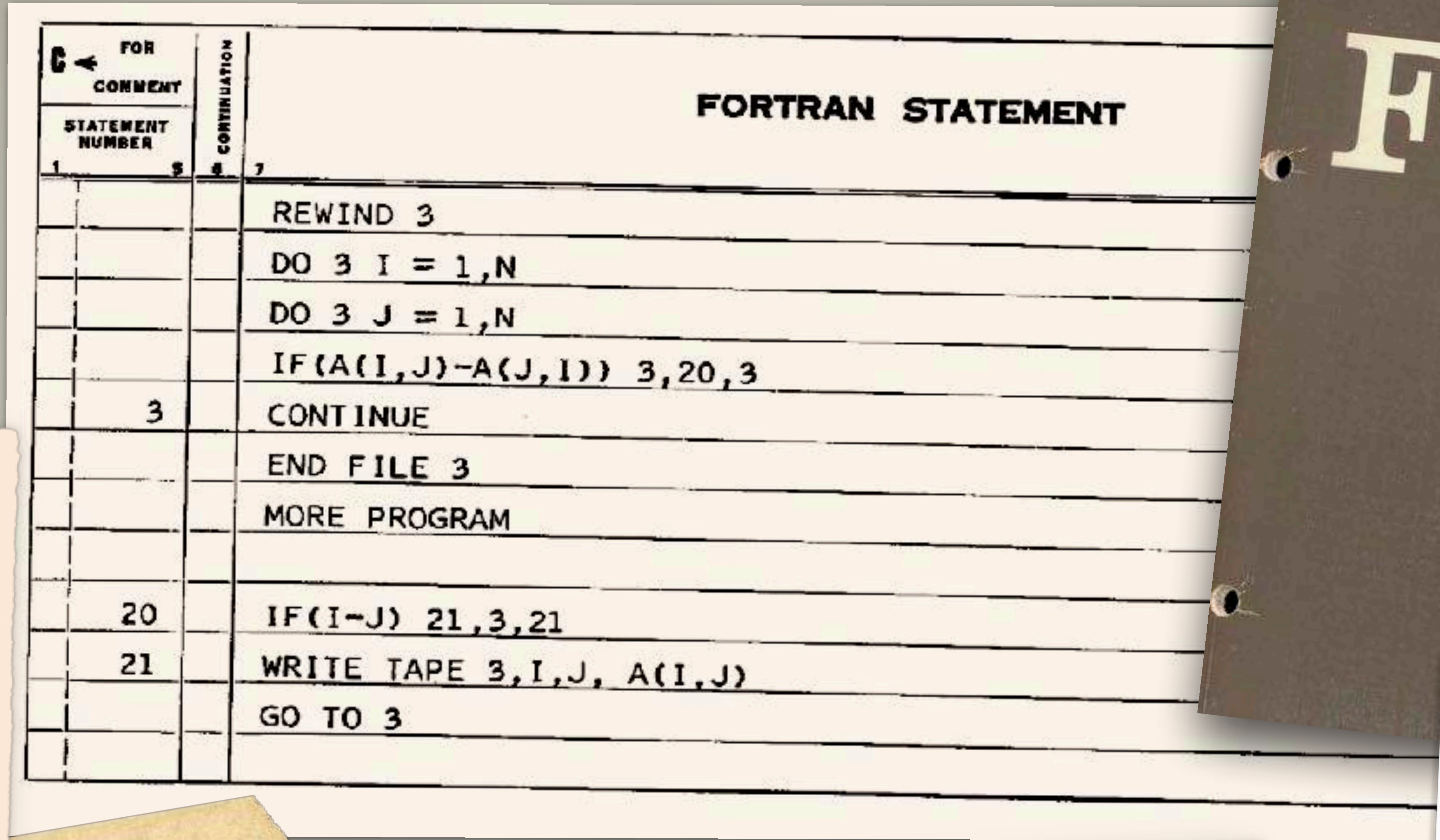
Early Programming Languages

FORTRAN, 1954

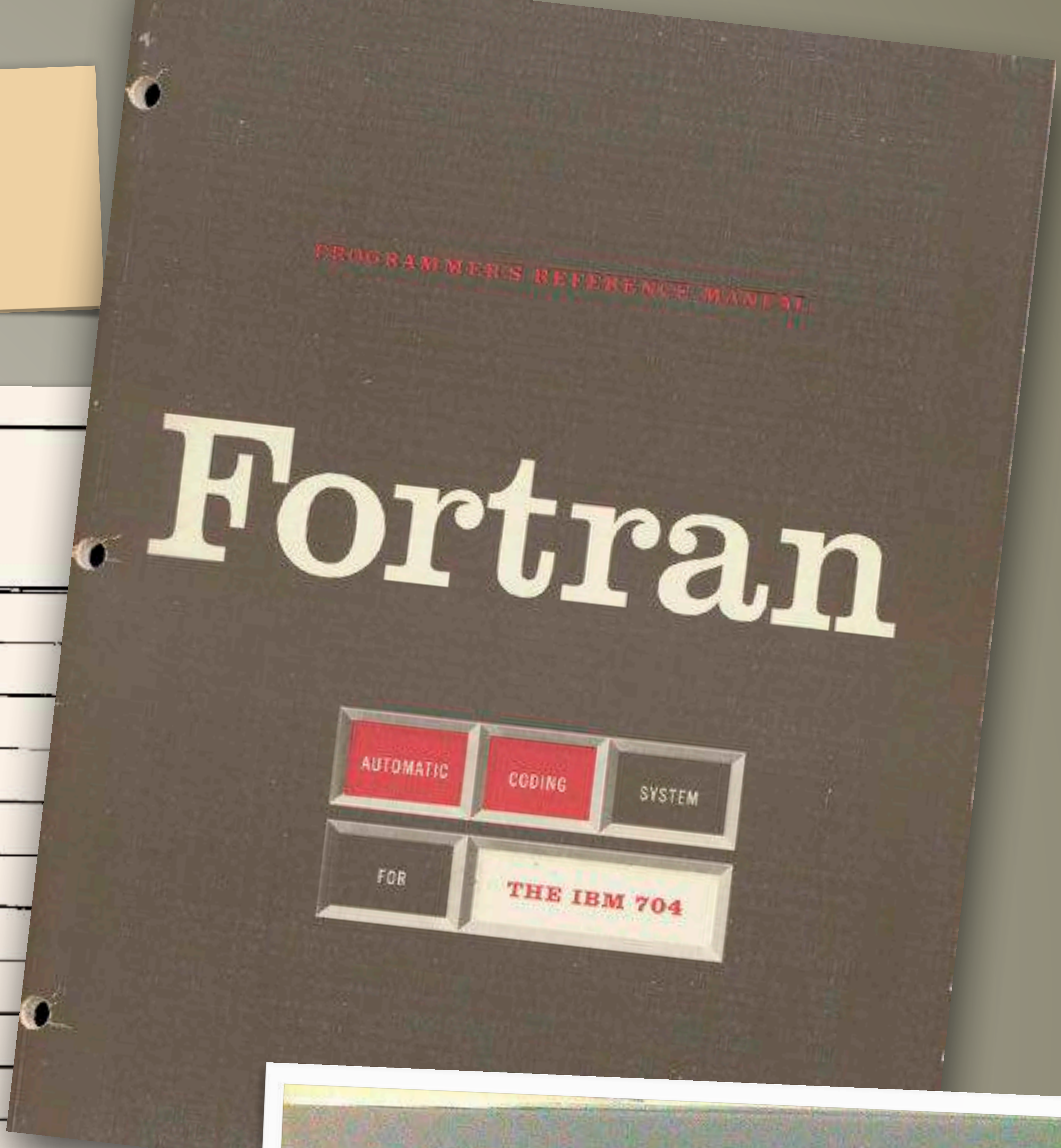


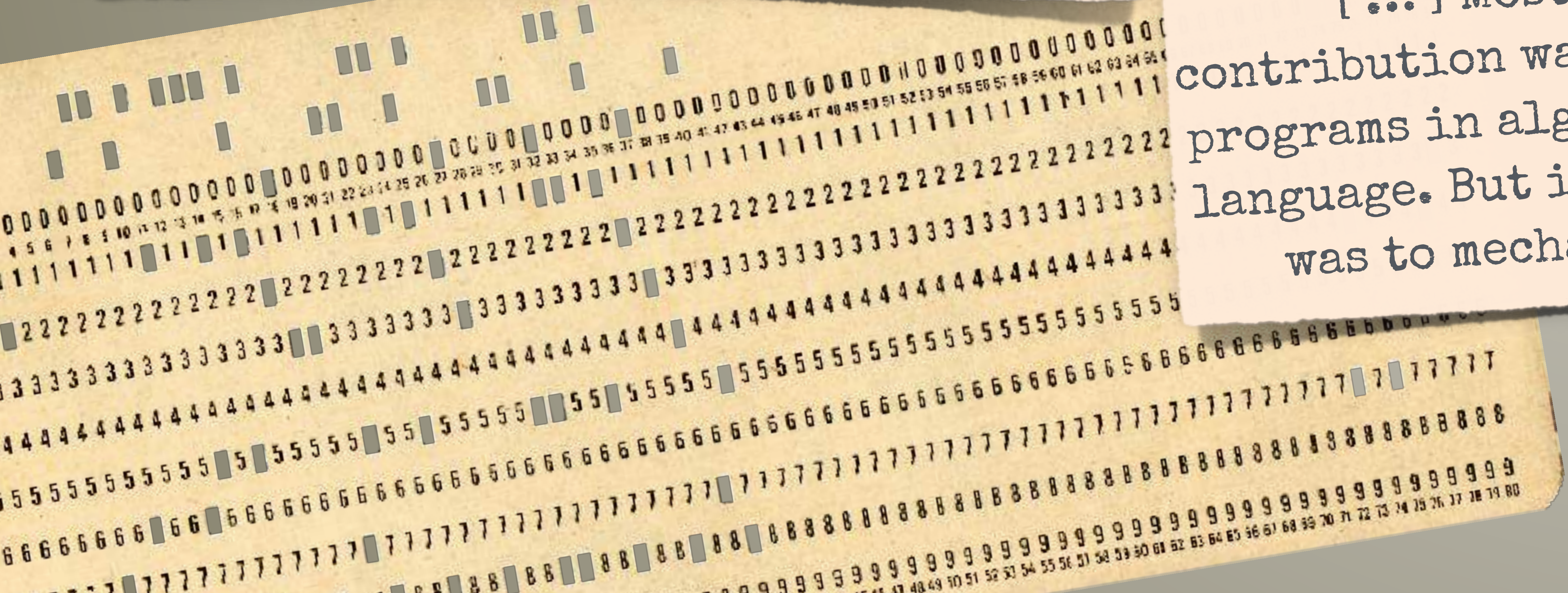
John Backus
1924 - 2007

Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs.



C ← FOR COMMENT / STATEMENT NUMBER	CONTINUATION	FORTRAN STATEMENT
		REWIND 3
		DO 3 I = 1,N
		DO 3 J = 1,N
		IF(A(I,J)-A(J,I)) 3,20,3
3		CONTINUE
		END FILE 3
		MORE PROGRAM
20		IF(I-J) 21,3,21
21		WRITE TAPE 3,I,J, A(I,J)
		GO TO 3

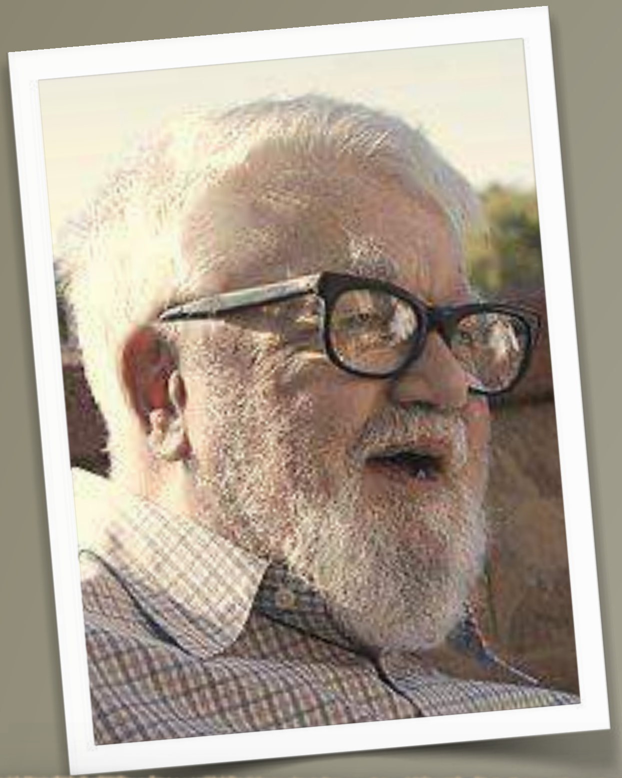




[...] Most people think FORTRAN's main contribution was to enable the programmer to write programs in algebraic formulas instead of machine language. But it isn't. What FORTRAN did primarily was to mechanize the organization of loops.

John Backus in "Think, IBM, July/August 1979"

LISP, 1958



John McCarthy

If $\mathcal{E}$ is a form in variables $x_1,\ldots,x_n$, then $\lambda((x_1,\ldots,x_n),\mathcal{E})$ will be taken to be the function of n variables whose value is determined by substituting the arguments for the variables $x_1,\ldots,x_n$ in that order in $\mathcal{E}$ and evaluating the resulting expression. For example, $\lambda((x,y),y^2+x)$ is a function of two variables, and $\lambda((x,y),y^2+x)(3,4) = 19$.

Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

$$(p_1 \rightarrow e_1,\ldots,p_n \rightarrow e_n)$$

where the p's are propositional expressions and the e's are expressions of any kind. It may be read, "If p_1 then e_1, otherwise if p_2 then $e_2,\ldots,$ otherwise if p_n then e_n," or "p_1 yields $e_1,\ldots,p_n$ yields e_n."

By using conditional expressions we can, without circularity, define functions by formulas in which the defined function occurs. For example, we write

$$n! = (n=0 \rightarrow 1, T \rightarrow n\cdot(n-1)!)$$

When we use this formula to evaluate 0! we get the answer 1;

g. Functions with Functions as Arguments

There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions. One such function is maplist[x;f] with an S-expression argument x and an argument f that is a function from S-expressions to S-expressions. We define

maplist[x;f] = [null[x] → NIL;T → cons[f[x];maplist[cdr[x];f]]]

An S-expression x that is not atomic is represented by a word, the address and decrement parts of which contain the locations of the subexpressions car[x] and cdr[x], respectively. In the list form of expressions the S-expression (A,(B,C),D) is represented by the list structure of Fig. 2.

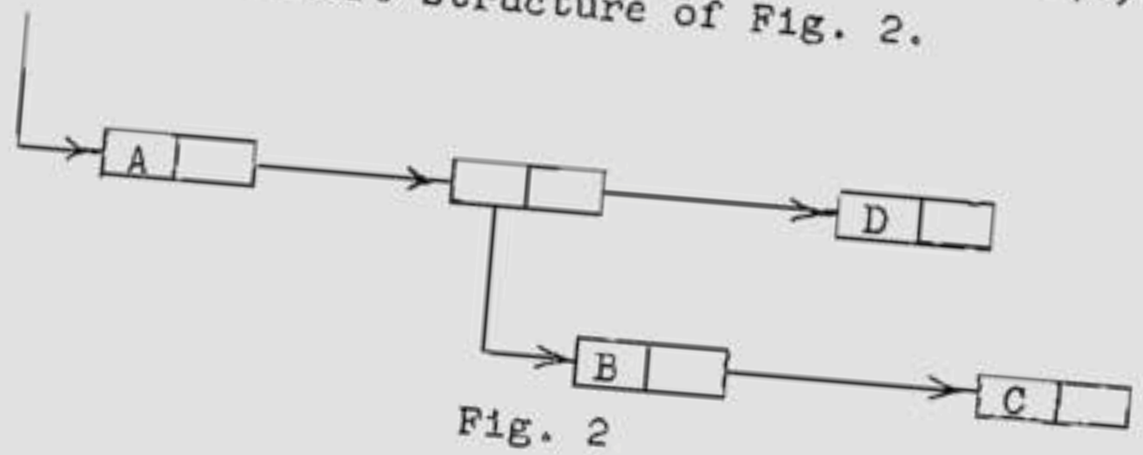


Fig. 2

When a list structure is regarded as representing a list, we see that each term of the list occupies the address part of a word, the decrement part of which points to the word containing the next term, while the last word has NIL in its decrement. The dot notation, e.g. (A·B), which is discussed in Section 2.2 is not allowed in LISP I; all lists and sublists must end with NIL.

Features of Lisp

- Based on lambda calculus
- Conditionals
- Recursive functions
- List processing (car, cdr, cons)
- Higher-order functions

LISP I

PROGRAMMER'S MANUAL

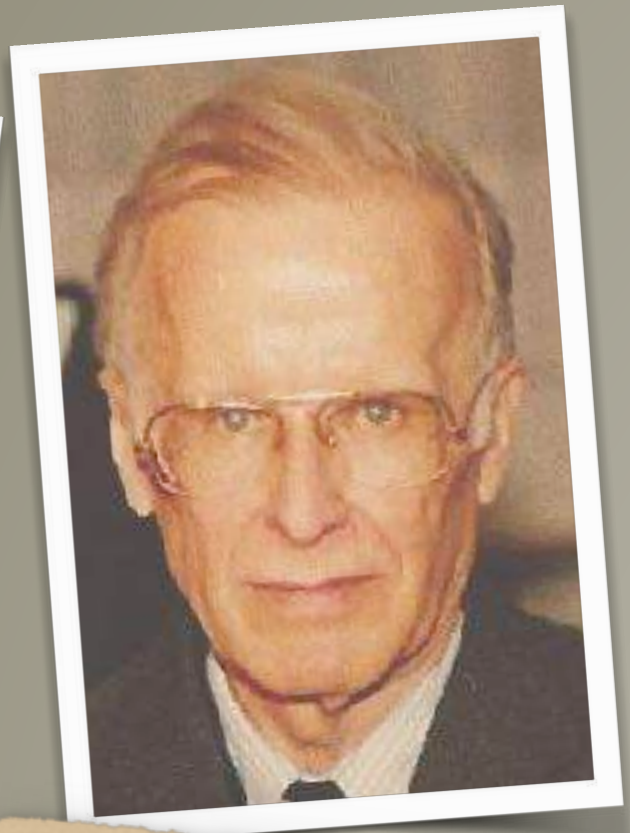
March 1, 1960

Artificial Intelligence Group

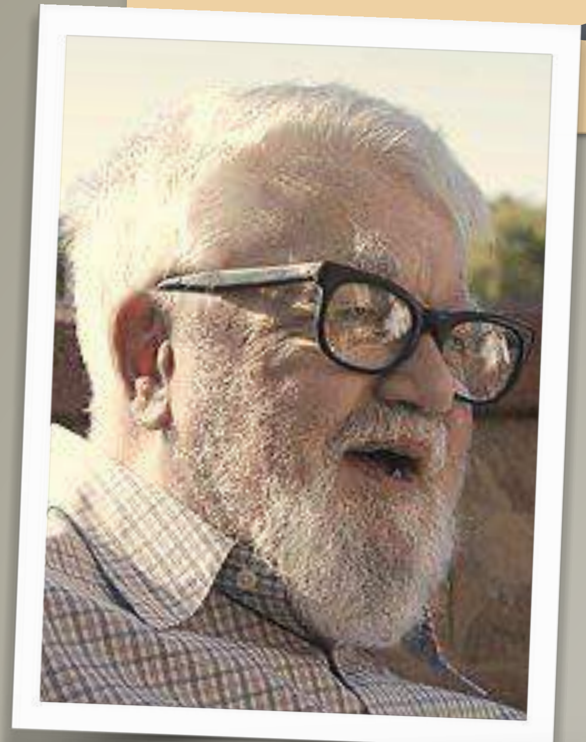
J. McCarthy
R. Brayton
D. Edwards
P. Fox
L. Hodes
D. Luckham
K. Maling
D. Park
S. Russell

Algol 1958

Peter Naur
1928 - 2016

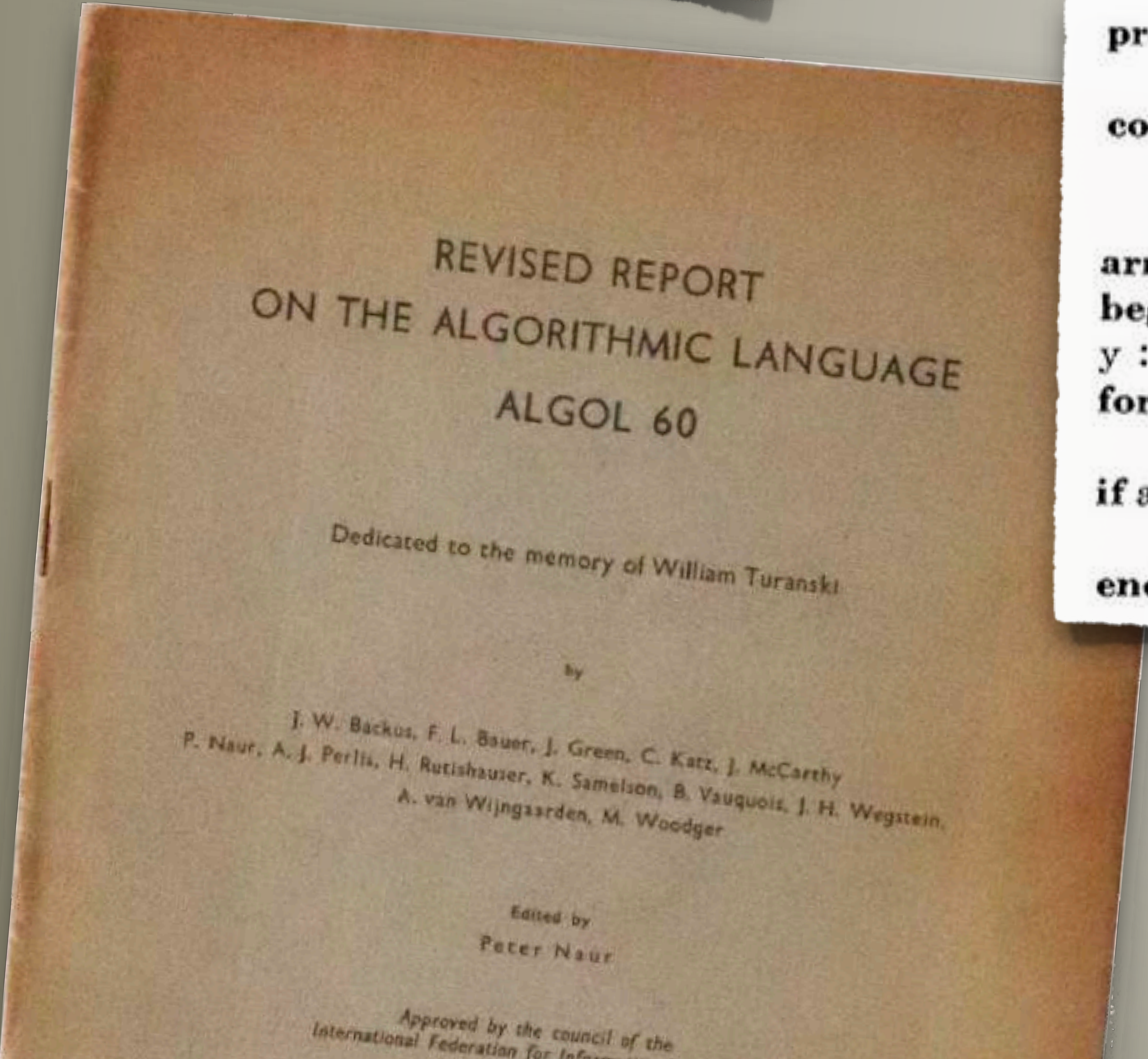


John Backus
1924 - 2007



John McCarthy
1927 - 2011

... and Green, Katz, Perlis, Rutishauser, Samelson, van Wijngaarden, Vauquois, Wegstein, and Woodger




```
procedure Absmax(a)size:(n,m)Result:(y)Subscripts:
    (i,k)   ;
comment The absolute greatest element of the matrix a,
    of size n by m is transferred to y, and the sub-
    scripts of this element to i and k   ;
array a   ;  integer n, m, i, k   ;  real y   ;
begin integer p, q   ;
y := 0   ;
for p := 1 step 1 until n do for q := 1 step 1 until
    m do
if abs(a[p,q]) > y then begin y: = abs(a[p,q])   ;  i: = p   ;
    k: = q
end end Absmax
```

Algol

- Iteration and recursion*
- Distinct assignment (:=) and equality (=)
- Code blocks and lexical scope
- Nested procedure, and procedure passing
- BNF
- Call-by-value and call-by-name

"... a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors" – Tony Hoare, 1974, in Hints on Programming Language Design

Algol and Lambdas, 1965

Peter Landin
1930 - 2009

Applicative expressions (AE) =
Syntactic sugar for lambda expressions
Imperative AE (IAE) =
AE + assignment operation (IAE)

SECD Machine
First abstract machine for evaluating lambda expressions

Here is a typical form of AE, presented both informally and formally.

let $a = A$	$\{\lambda(a, b, f).$
and $b = B$	$\{\lambda(g, h).$
and $f(x, y) = F$	$\{\lambda k.X\}\{\lambda(u, v).\lambda x.K\}\}$
let rec $g(x) = G$	$[Y\lambda(g, h).(\lambda x.G, \lambda(y, z).H]\}$
and $h(y, z) = H$	$[A, B, \lambda(x, y).F]$
let $k(u, v)(x) = K$	
X	

The only consideration in choosing between **let** and **where** will be the relative convenience of writing an auxiliary definition before or after the expression it qualifies.

1964

The mechanical evaluation of expressions

By P. J. Landin

This paper is a contribution to the "theory" of the activity of using computers. It s some forms of expression used in current programming languages can be modelled in λ-notation, and then describes a way of "interpreting" such expressions. This s method, of analyzing the things computer users write, that applies to many different orientations and to different phases of the activity of using a computer. Also a tec introduced by which the various composite information structures involved can be characterized in their essentials, without commitment to specific written or other represe

Introduction

The point of departure of this paper is the idea of a machine for evaluating schoolroom sums, such as

1. $(3 + 4)(5 + 6)(7 + 8)$
2. if $2^{19} < 3^{12}$ then ${}^{12}\sqrt{2}$ else ${}^{53}\sqrt{2}$
3. $\sqrt{\left(\frac{17 \cos \pi/17 - \sqrt{(1 - 17 \sin \pi/17)}}{17 \cos \pi/17 + \sqrt{(1 + 17 \sin \pi/17)}}\right)}$

Any experienced computer user knows that his activity scarcely resembles giving a machine a numerical expression and waiting for the answer. He is involved with flow diagrams, with replacement and sequencing, with programs, data and jobs, and with input and output. There are good reasons why current information processing

is written explicitly and prefixed to its operand(each operand (or operand-list) is enclosed in br e.g.

$$f(a, +(\times(2, b), 3)).$$

This notation is a sort of standard notation in all the expressions in this paper could (with som of legibility) be rendered.

The following remarks about applicative structure will be illustrated by examples in which an expression is written in two ways: on the left in some notation whose applicative structure is being discussed, and on the right in a form that displays the applicative structure more explicitly, e.g.

TABLE 1

ALGOL 60	IAEs
Identifiers, operator symbols, also some special words and configurations	Identifiers
Local identifiers	Variables bound in a λ-expression occurring as operator
Formal parameters	Variables bound in a λ-expression occurring as operand
Function designator, subscripted variable, and procedure statement	Operator/operand combination
Procedure	λ-expression
Actual parameter called by name	λ ( )-expression

1965

Standards

S. GORN, Editor; R. W. BEMER, Asst. Editor, Glossary & Term
E. LOHSE, Asst. Editor, Information Intercha
R. V. SMITH, Asst. Editor, Programming Lan

A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I*

By P. J. Landin†

This paper describes how some of the semantics of ALGOL 60 can be formalized by establishing a correspondence between expressions of ALGOL 60 and expressions in a modified form of Church's λ-notation. First a model for computer languages and computer behavior is described, based on the notions of functional application and functional abstraction, but also having analogues for imperative language features. Then this model is used as an "abstract object language" into which ALGOL 60 is mapped. Many of ALGOL 60's features emerge as particular arrangements of a small number of structural rules, suggesting new classifications and generalizations.

The correspondence is first described informally, mainly by illustrations. The second part of the paper gives a formal description, i.e. an "abstract compiler" into the "abstract object language." This is itself presented in a "purely functional" notation, that is one using only application and abstraction.

Introduction

Anyone familiar with both Church's λ-calculi [7]) and ALGOL 60 [8] will have noticed a super semblance between the way variables tie up wit in a nest of λ-expressions, and the way identifie with the headings in a nest of procedures and bloc may also have observed that in, say

$$\{\lambda f.f(a) + f(b)\}[\lambda x.x^2 + px + q]$$

the two λ-expressions, i.e. the operator and th play roughly the roles of block-body and declaration, respectively. The present paper ex resemblance in some detail.

The presentation falls into four sections. Th tion, following this introduction, gives some for examining the correspondence. The second scribes an abstract language based on Church This abstract language is a development of the system presented in [3] and some acquaintan paper (hereinafter referred to as [MEE]) is as The third section describes informally, mai trations, a correspondence between expressio 60 and expressions of the abstract languag section formalizes this correspondence; it fi a sort of "abstract ALGOL 60" and then prese tions that map expressions of abstract ALGO the one hand, ALGOL 60 texts, and on the expressions of the abstract language.

Motivation

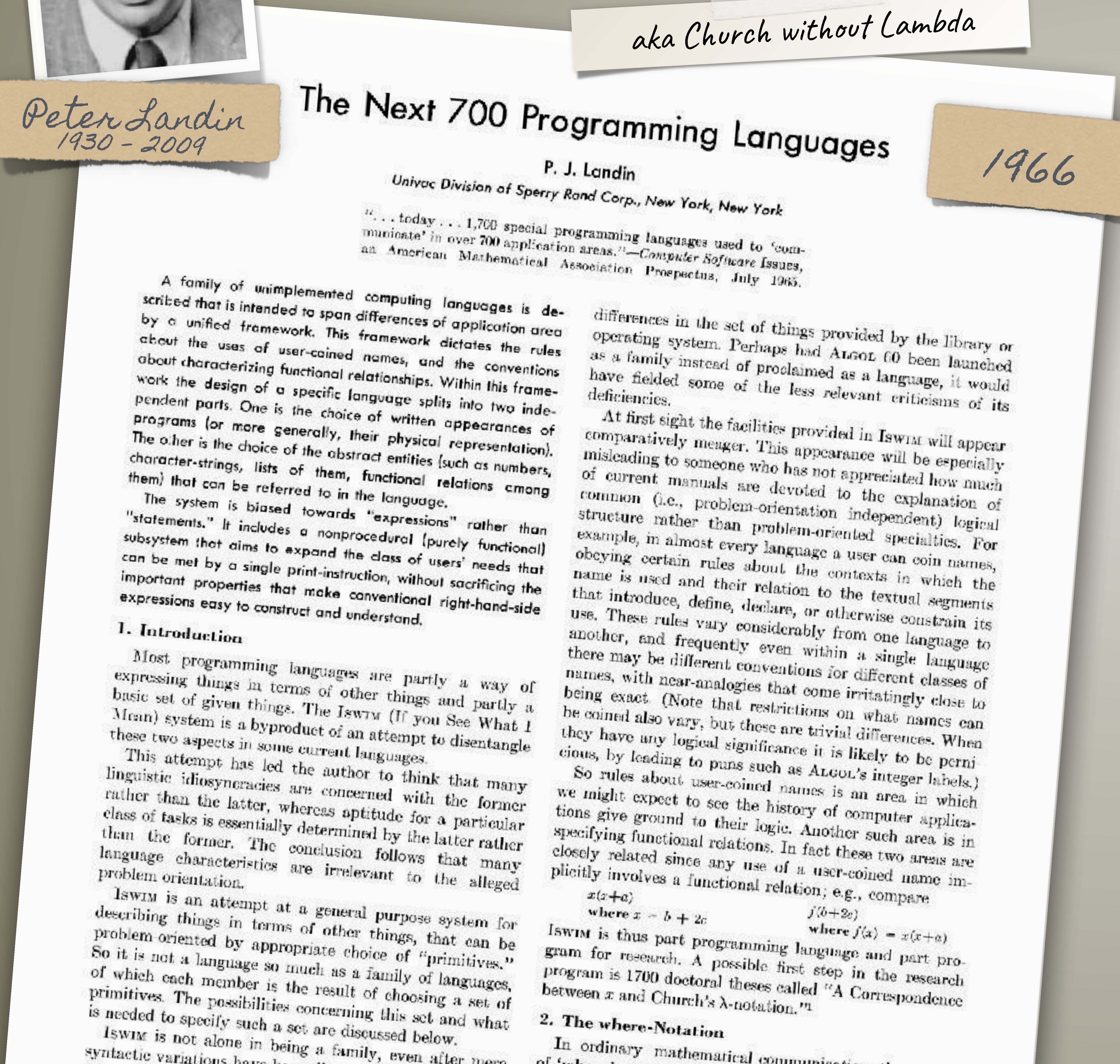
It seems possible that the corresponden the basis of a formal description of the sema 60. As presented here it reduces the proble

ISWIM, 1966

(If you See What I Mean)

Peter Landin
1930 - 2009

aka Church without Lambda



The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software Issues*, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is de-
scribed that is intended to span differences of application area
by a unified framework. This framework dictates the rules
about the uses of user-coined names, and the conventions
about characterizing functional relationships. Within this frame-
work the design of a specific language splits into two inde-
pendent parts. One is the choice of written appearances of
programs (or more generally, their physical representation).
The other is the choice of the abstract entities (such as numbers,
character-strings, lists of them, functional relations among
them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of
expressing things in terms of other things and partly a
basic set of given things. The ISWIM (If you See What I
Mean) system is a byproduct of an attempt to disentangle
these two aspects in some current languages.

This attempt has led the author to think that many
linguistic idiosyncracies are concerned with the former
rather than the latter, whereas aptitude for a particular
class of tasks is essentially determined by the latter rather
than the former. The conclusion follows that many
language characteristics are irrelevant to the alleged
problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after mere syntactic variations have been

differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be perni cious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(x+a)$
where $x = b + 2c$

$f(b+2c)$
where $f(x) = x(x+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication

1966

ISWIM

- Mathematical notation for functions, using **let** and **where** clauses
- Indentation to separate clauses (offside rule)
- A distinction between physical and logical languages
- Interpretation through layers of abstraction linked by syntax, grammar, and denotation
- Garbage collection
- Equivalences between terms
- No explicit sequencing
- Algebraic datatypes

Church without Lambda

$$(u - 1)(u + 2) \quad \{\lambda u.(u - 1)(u + 2)\}[7 - 3]$$
where $u = 7 - 3.$

$$u(u + 1) - v(v + 1) \quad \{\lambda(u, v).u(u + 1) - v(v + 1)\}$$
where $u = 2p + q \quad [2p + q, p - 2q]$
and $v = p - 2q.$

$f(7 - 3) \quad \{\lambda f.f(7 - 3)\}$
where rec $f(n) =$ **if** $n = 0$ **then** 1 **else** $nf(n - 1)$

$[Y\lambda f.\lambda n.$**if** $n = 0$ **then** 1 **else** $nf(n - 1)].$

where $n = round(n)$
instead of the specification

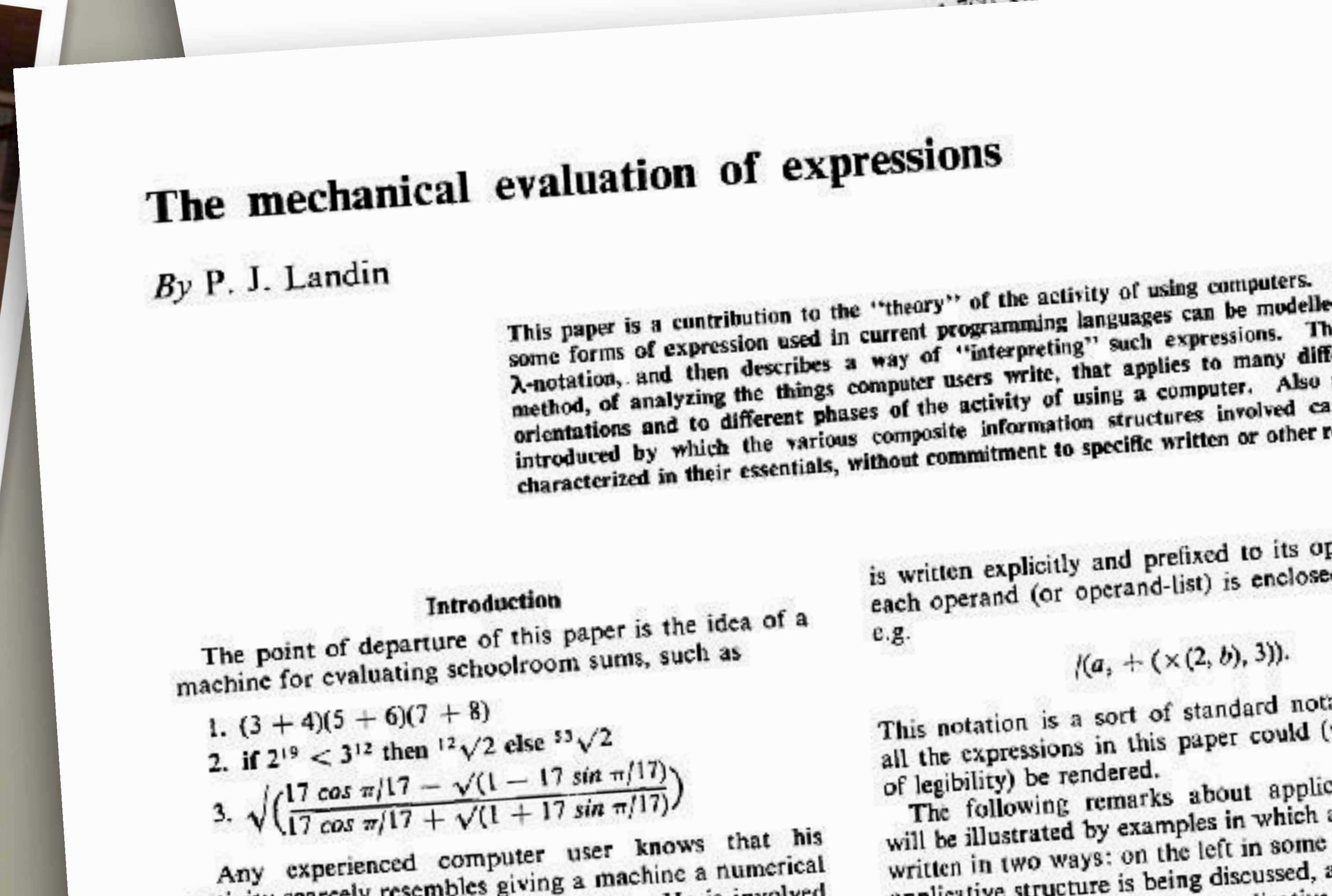
where notation

- More convenient that lambda terms
- Indentation sensitive
- Multiple and recursive definitions
- Used for types

Peter Naur
1928 - 2016

Regarding indentation, in many ways I am in sympathy with this, but [..] you would regret it because of the kind of rearrangement of manuscripts done in printing

The Next 700 Programmin
P. J. Landin
Univac Division of Sperry Rand Corp., New Y



The mechanical evaluation of expressions

By P. J. Landin

Introduction

The point of departure of this paper is the idea of a machine for evaluating schoolroom sums, such as

1. $(3 + 4)(5 + 6)(7 + 8)$
2. if $2^{19} < 3^{12}$ then $^{12}\sqrt{2}$ else $^{53}\sqrt{2}$

Language Abstraction

The "physical/logical" terminology is often used to distinguish features that are a fortuitous consequence of physical conditions from features that are in some sense more essential. This idea is carried further by making a similar distinction among the "more essential" features. In fact ISWIM is presented here as a four-level concept comprising the following:

Physical ISWIM

tokenize

Logical ISWIM

parse

Abstract ISWIM

interpret

Applicative Expression

evaluate

Value

f (b + 2c)
where f(x) = x(x+a)

```
f(b + 2c)
  where f(x) = x(x+a)
```

1/10th of the paper is devoted to the idea of abstraction layers for defining a language!

Today this is regarded as the standard means of giving language semantics

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

The mechanical evaluation of expressions

By P. J. Landin

1964

Equivalences

Enabling equational reasoning featured heavily in this paper

(1) the extent to which a subexpression can be replaced by an equivalent subexpression without disturbing the equivalence class of the whole expression. Without this group the other rules would be applicable only to complete expressions, not to subexpressions.
(2) user-coined names, i.e., in definitions and, in particular, function definitions.
(3) built-in entities implicit in special forms of expression. The only instances of this in ISWIM are conditional expressions, listings and self-referential definitions.
(4) named entities added in any specific problem-orientation of ISWIM.

(1): subexpressions

(μ) If $L \equiv L'$ then $L\ (M) \equiv L'\ (M)$

(ν) If $M \equiv M'$ then $L\ (M) \equiv L\ (M')$

(2): definitions

$(\textbf{let})$ $\textbf{let}\ x = M;\ L \equiv L\ \textbf{where}\ x = M$

(3): built-in expressions

$(\rightarrow)$ $\textbf{true} \rightarrow M;\ N \equiv M$

$(\rightarrow')$ $\textbf{false} \rightarrow M;\ N \equiv N$

(4): primitives

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what

A problem-orientation of ISWIM can be characterized by additional axioms. In the simplest case such an axiom is an ISWIM definition. The resulting modification is called a "definitional extension" of the original system.

Denotation

The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack. In particular, (a) each expression has a nesting subexpression structure, (b) each subexpression denotes something (usually a number, truth value or numerical function), (c) the thing an expression denotes, i.e., its "value", depends only on the *values* of its sub-expressions, not on other properties of them.

It is these properties, and crucially (c), that explains why such expressions are easier to construct and under-stand. Thus it is (c) that lies behind the evolutionary trend towards "bigger righthand sides" in place of strings of small, explicitly sequenced assignments and jumps.

• The importance of denotational semantics was clearly highlighted

$$\mathcal{V}[\nu_0 \oplus \nu_1] = \mathcal{V}[\nu_0] + \mathcal{V}[\nu_1]$$

$$\mathcal{C}[\varepsilon \to \gamma_0, \gamma_1] = Cond(\mathcal{C}[\gamma_0], \mathcal{C}[\gamma_1]) * \mathcal{E}[\varepsilon],$$
$$\mathcal{C}[\gamma_0 ; \gamma_1] = \mathcal{C}[\gamma_1] \circ \mathcal{C}[\gamma_0].$$

Dana Scott

Christopher Strachey
1916 – 1975

1971

Toward a Mathematical Semantics for Computer Languages

by

Dana Scott
Princeton University

and

Christopher Strachey
Oxford University

Technical Monograph PRG-6

functional

Alonzo Church

ISWIM brings into sharp relief some of the distinctions that the author thinks are intended by such adjectives as procedural, nonprocedural, algorithmic, heuristic, imperative, declarative, functional, descriptive. Here is a suggested classification, and one new word.

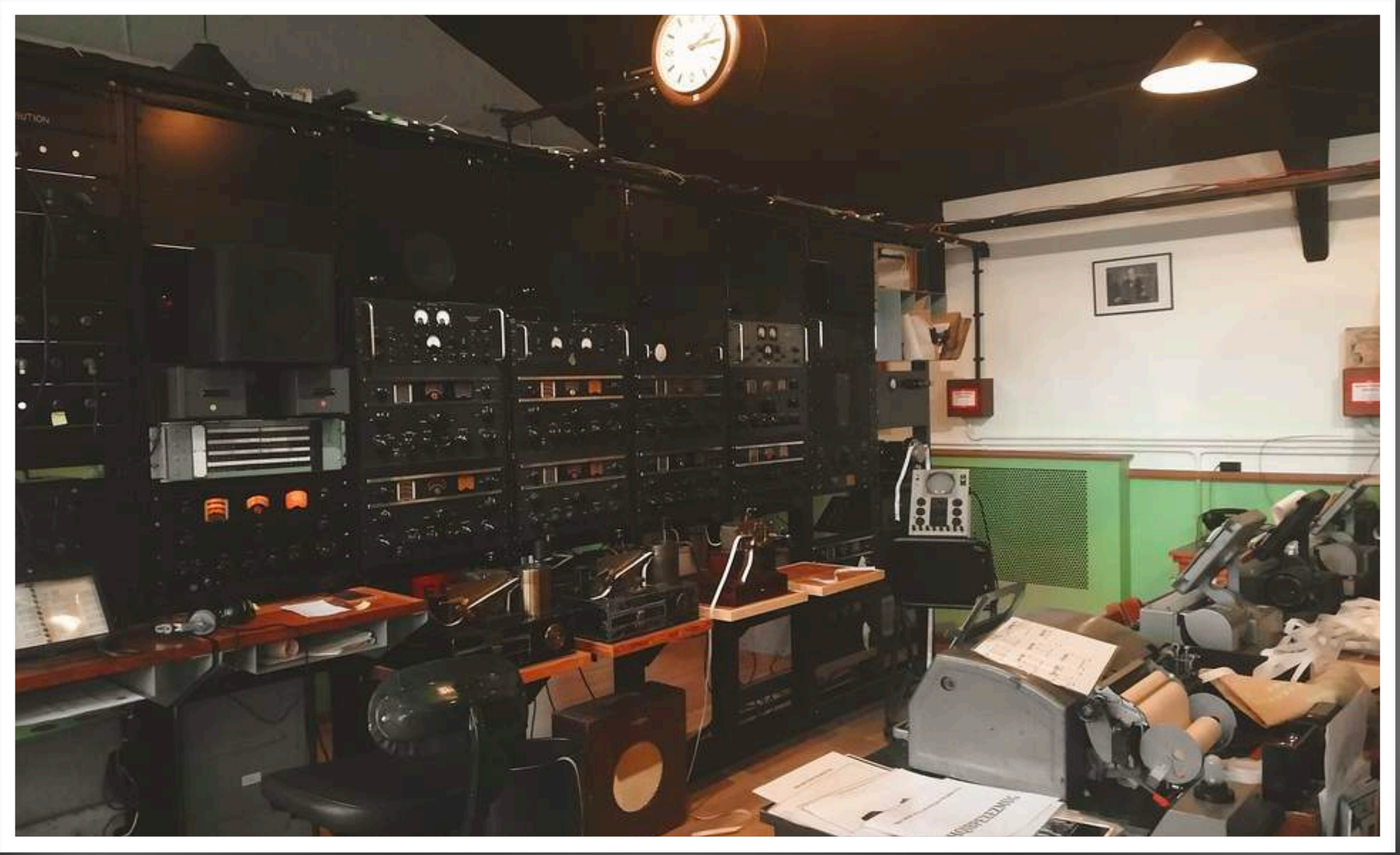
Peter Landin

denotative!

```
> imperative
```

Alan Turing

The Next 700 ~~Programming~~ Domain Specific Languages



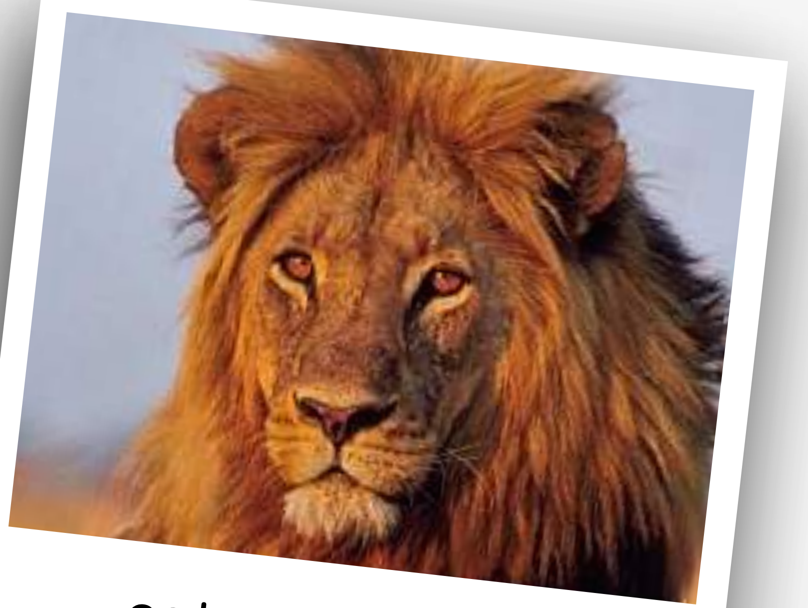
Datastructures and Recursion Schemes

Lists can be evaluated by folding their structure: cons and nil are replaced with a semantics of what to do

This technique generalises to many tree-like datatypes

There are many variations on this theme, depending on mutual recursion, recursion with parameters, recursion with historic context, and many more

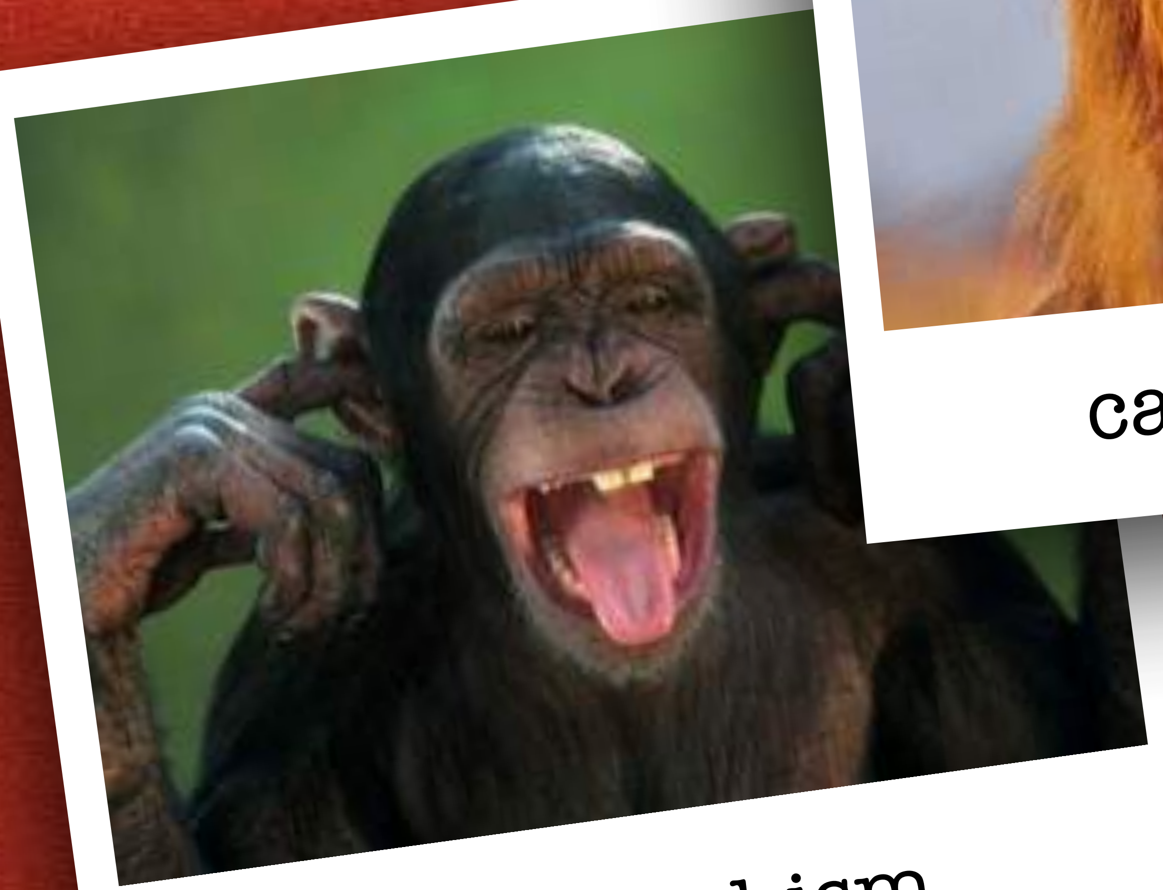
```
               [2,  7,  1,  8]

foldr (+) 0 (2 : 7 : 1 : 8 : [])

          =  2 + 7 + 1 + 8 + 0  = 18
```



catamorphism

catamorphism

anamorphism



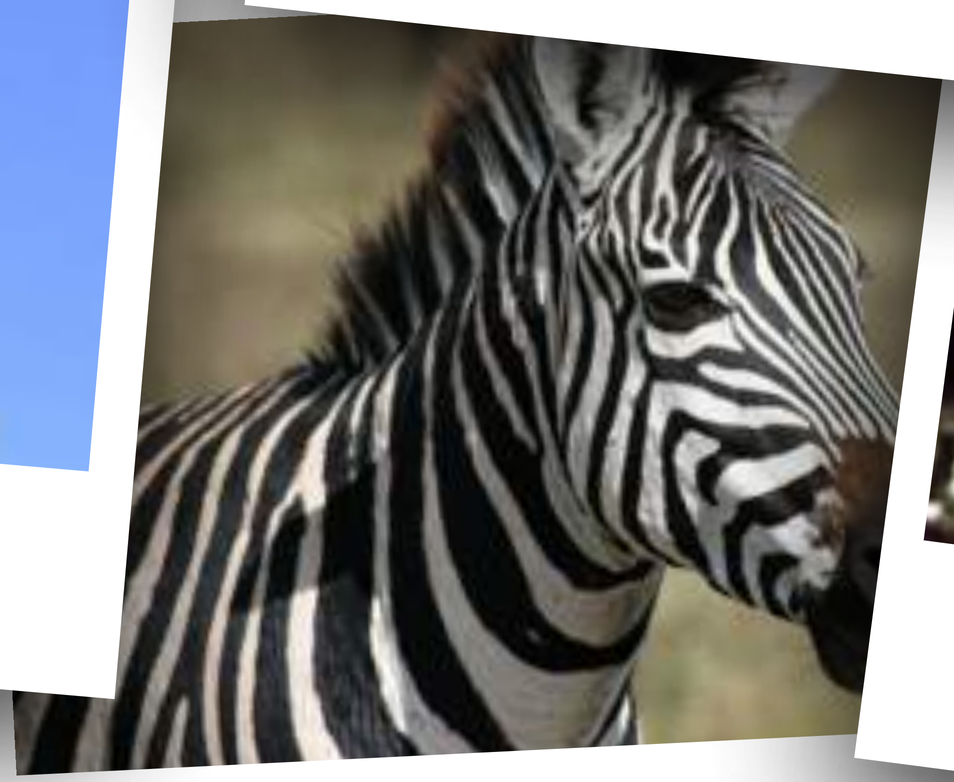
apomorphism

paramorphism

histomorphism

futumorphism

mutumorphism



zygomorphism

dynamorphism

$Free^M \dashv U^M$

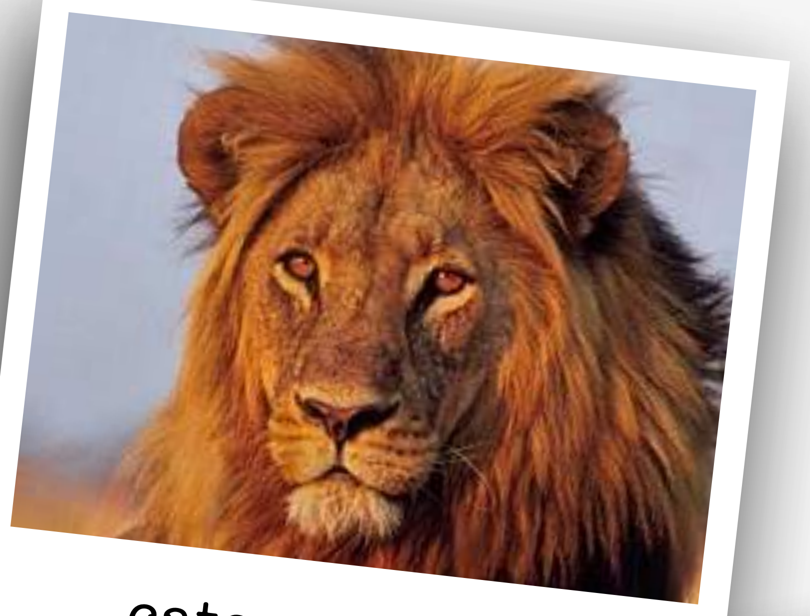
Datastructures and Recursion Schemes

Lists can be evaluated by folding their structure: cons and nil are replaced with a semantics of what to do

This technique generalises to many tree-like datatypes

There are many variations on this theme, depending on mutual recursion, recursion with parameters, recursion with historic context, and many more

```
[2,   7,   1,   8]

foldr (+) 0 (2 : 7 : 1 : 8 : [])

=   2 + 7 + 1 + 8 + 0 = 18
```



catamorphism

2013

Unifying Structured Recursion Schemes

Ralf Hinze   Nicolas Wu   Jeremy Gibbons
Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England
{ralf.hinze,nicolas.wu,jeremy.gibbons}@cs.ox.ac.uk

2015

Conjugate Hylomorphisms

Or: The Mother of All Structured Recursion Schemes

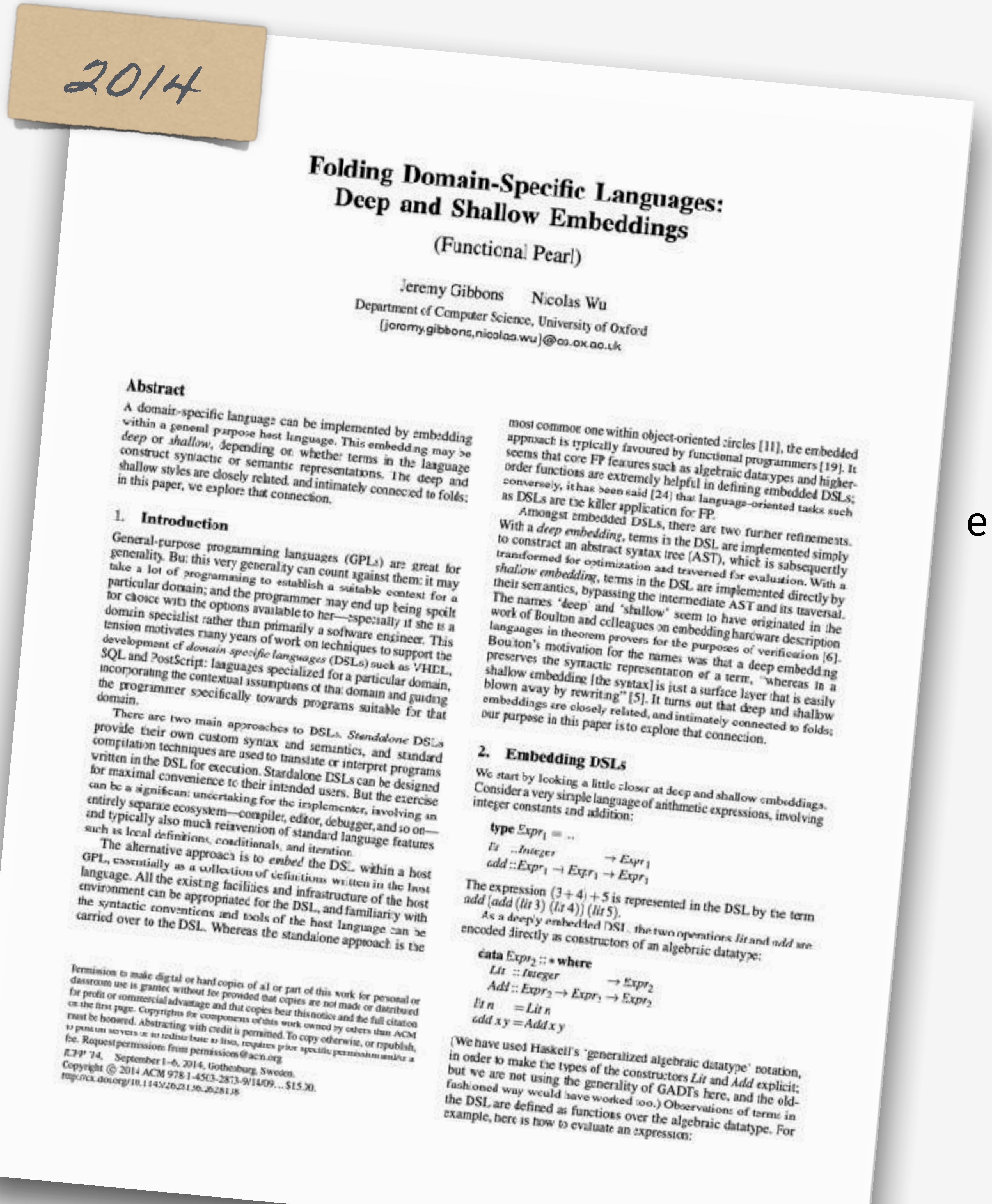
Ralf Hinze   Nicolas Wu   Jeremy Gibbons
Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England
{ralf.hinze,nicolas.wu,jeremy.gibbons}@cs.ox.ac.uk

2022

Fantastic Morphisms and Where to Find Them
A Guide to Recursion Schemes

Zhixuan Yang and Nicolas Wu
Imperial College London, United Kingdom
{s.yang20,n.wu}@imperial.ac.uk

Domain-Specific Languages



syntax tree + recursion scheme

=

language + denotational semantics

```
data Expr where
  Add :: Expr → Expr → Expr
  Var :: String → Expr
```

$$[\![x + y]\!] = [\![x]\!] \oplus [\![y]\!]$$

```
eval alg  (Add (Var x) (Var y)) = eval alg (Var x) + eval alg (Var y)
```

```
eval :: (ExprF a → a) → Expr → a
eval alg (Op m n) = alg (OpF (eval alg m) (eval alg n))
eval alg (Var x)  = alg (VarF x)
```

```
data ExprF k where
  OpF  :: k → k → k
  VarF :: String → k
```

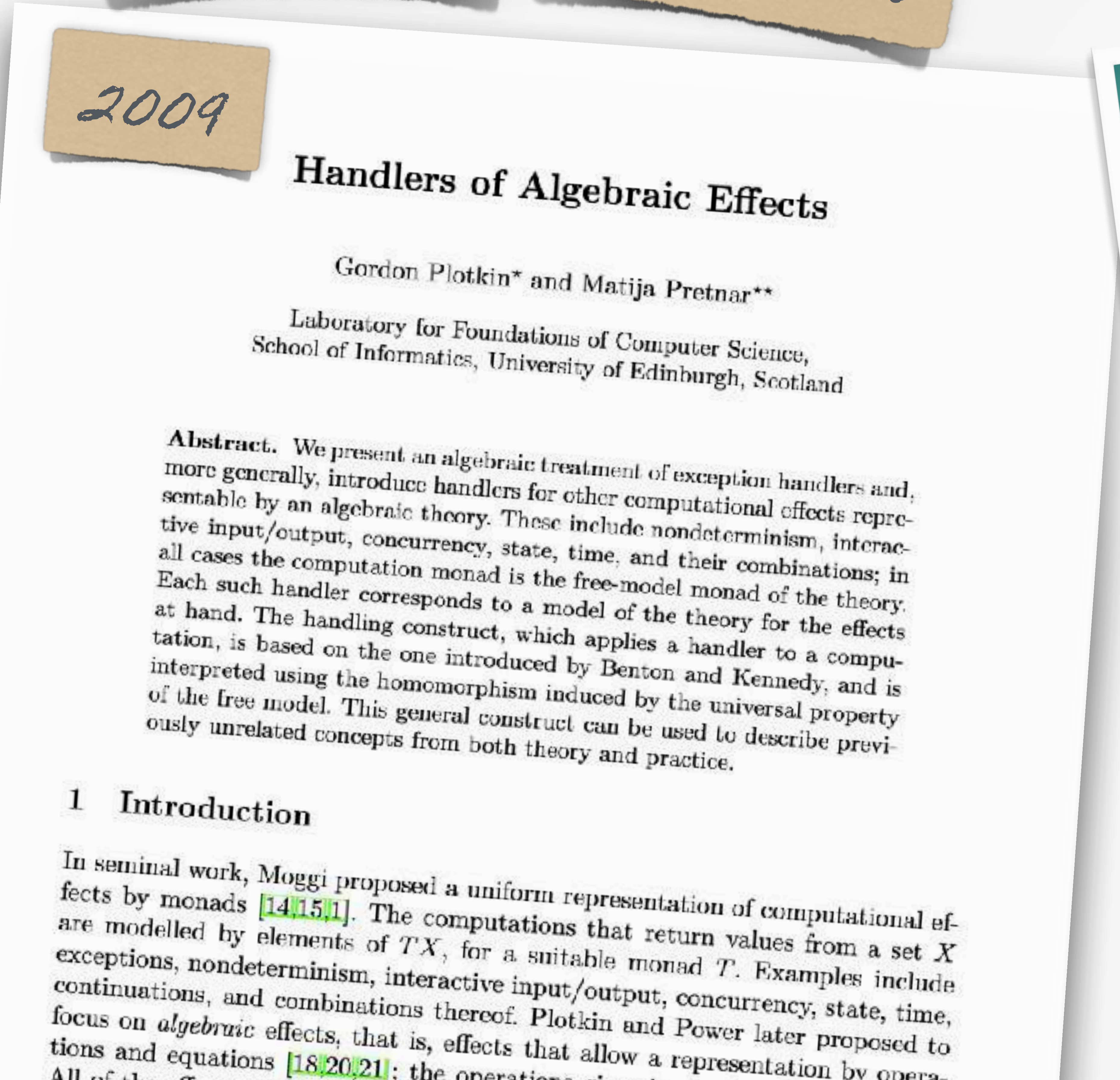
```
alg :: ExprF a → a
alg (OpF a b) = a + b
alg (VarF x)  = var x
```

Algebraic Effect Handlers

Gordon Plotkin

Matija Pretnar

2009



Handlers of Algebraic Effects

Gordon Plotkin* and Matija Pretnar**

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, Scotland

Abstract. We present an algebraic treatment of exception handlers and, more generally, introduce handlers for other computational effects representable by an algebraic theory. These include nondeterminism, interactive input/output, concurrency, state, time, and their combinations; in all cases the computation monad is the free-model monad of the theory. Each such handler corresponds to a model of the theory for the effects at hand. The handling construct, which applies a handler to a computation, is based on the one introduced by Benton and Kennedy, and is interpreted using the homomorphism induced by the universal property of the free model. This general construct can be used to describe previously unrelated concepts from both theory and practice.

1 Introduction

In seminal work, Moggi proposed a uniform representation of computational effects by monads [14,15,1]. The computations that return values from a set X are modelled by elements of TX, for a suitable monad T. Examples include exceptions, nondeterminism, interactive input/output, concurrency, state, time, continuations, and combinations thereof. Plotkin and Power later proposed to focus on *algebraic* effects, that is, effects that allow a representation by operations and equations [18,20,21]; the operations ...

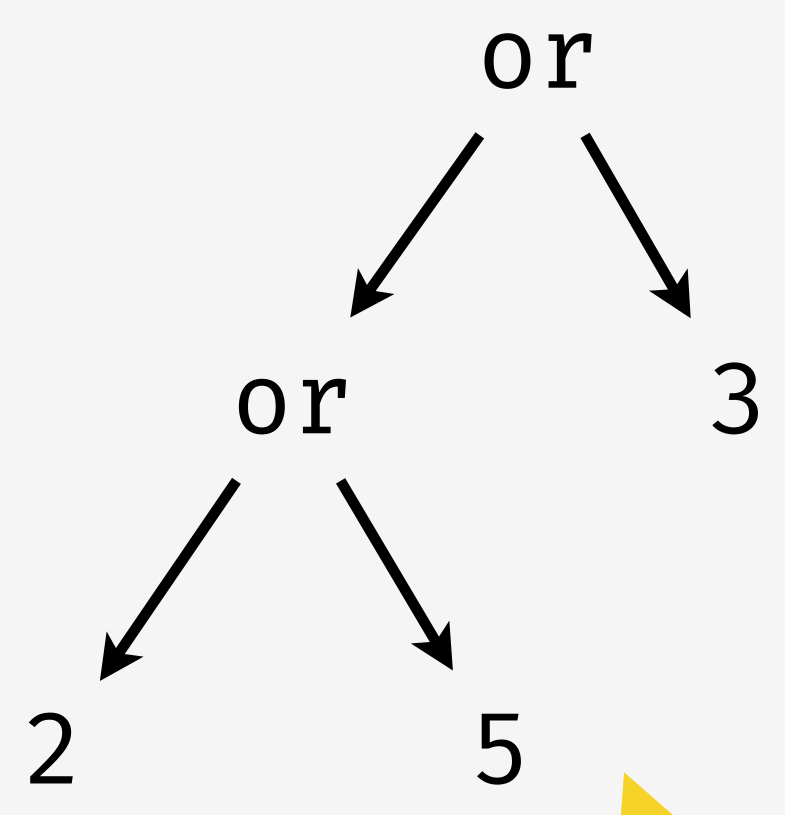
Syntax

Syntax represented by the free monad for a functor that provides a signature

Semantics

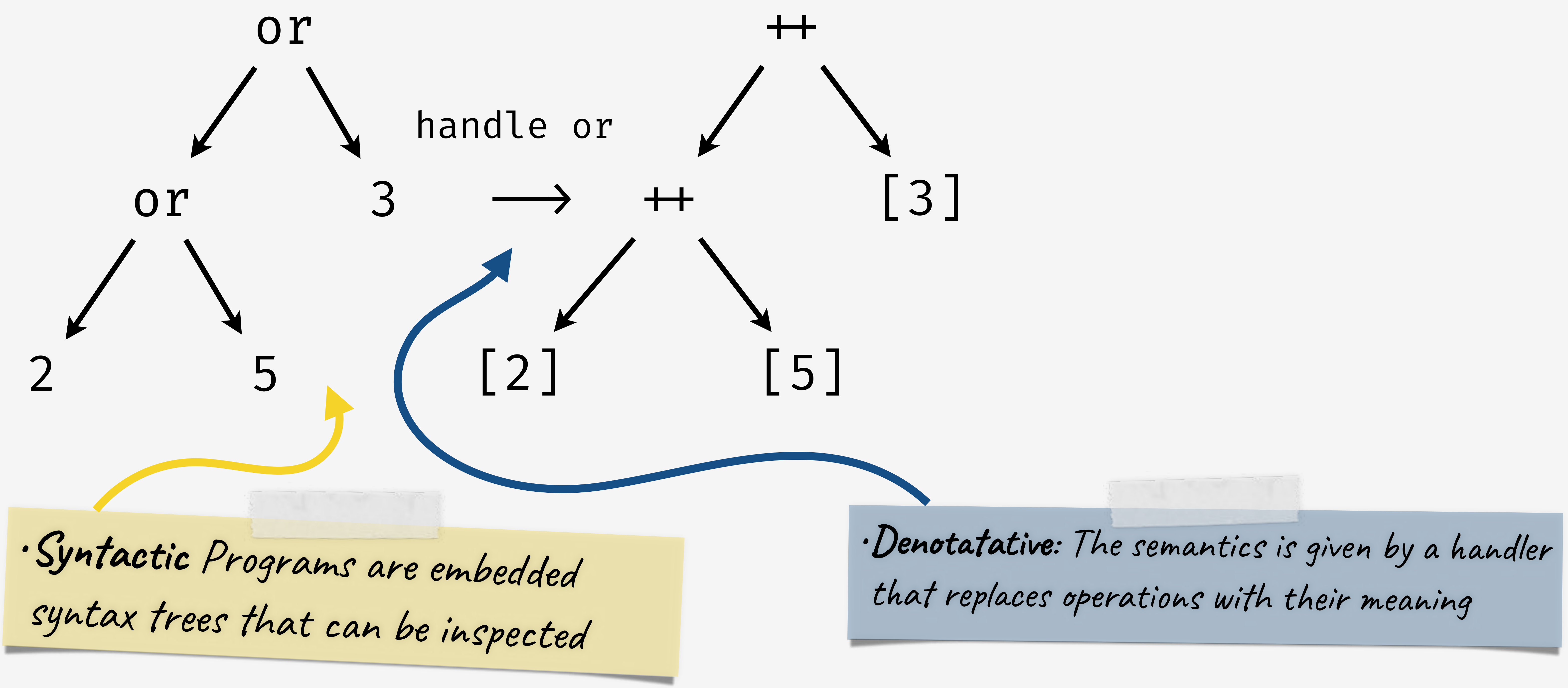
Semantics often in terms of a fold over the free monad

Algebraic Effect Handlers



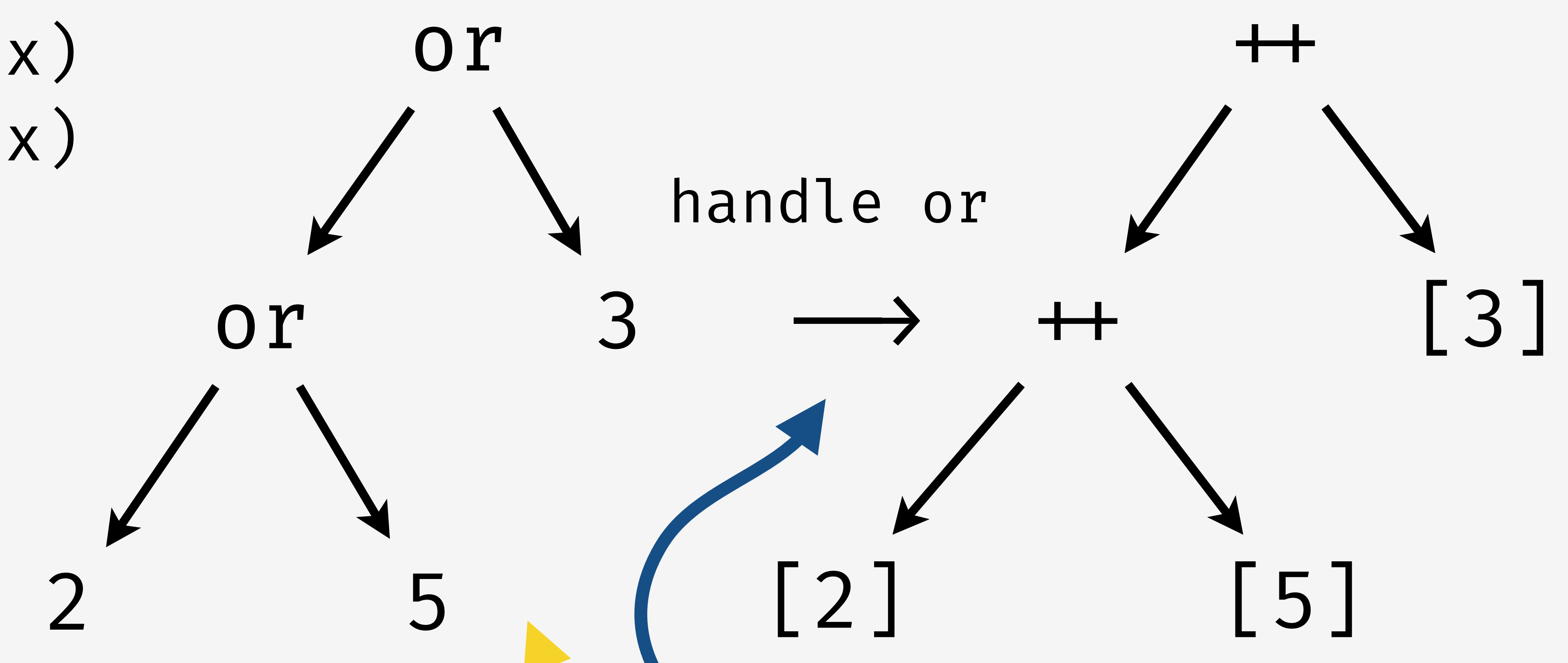
· *Syntactic* Programs are embedded syntax trees that can be inspected

Algebraic Effect Handlers



Algebraic Effect Handlers

```
do x ← or p q    = do or (do x ← p; k x)
   k x                   (do x ← q; k x)
```



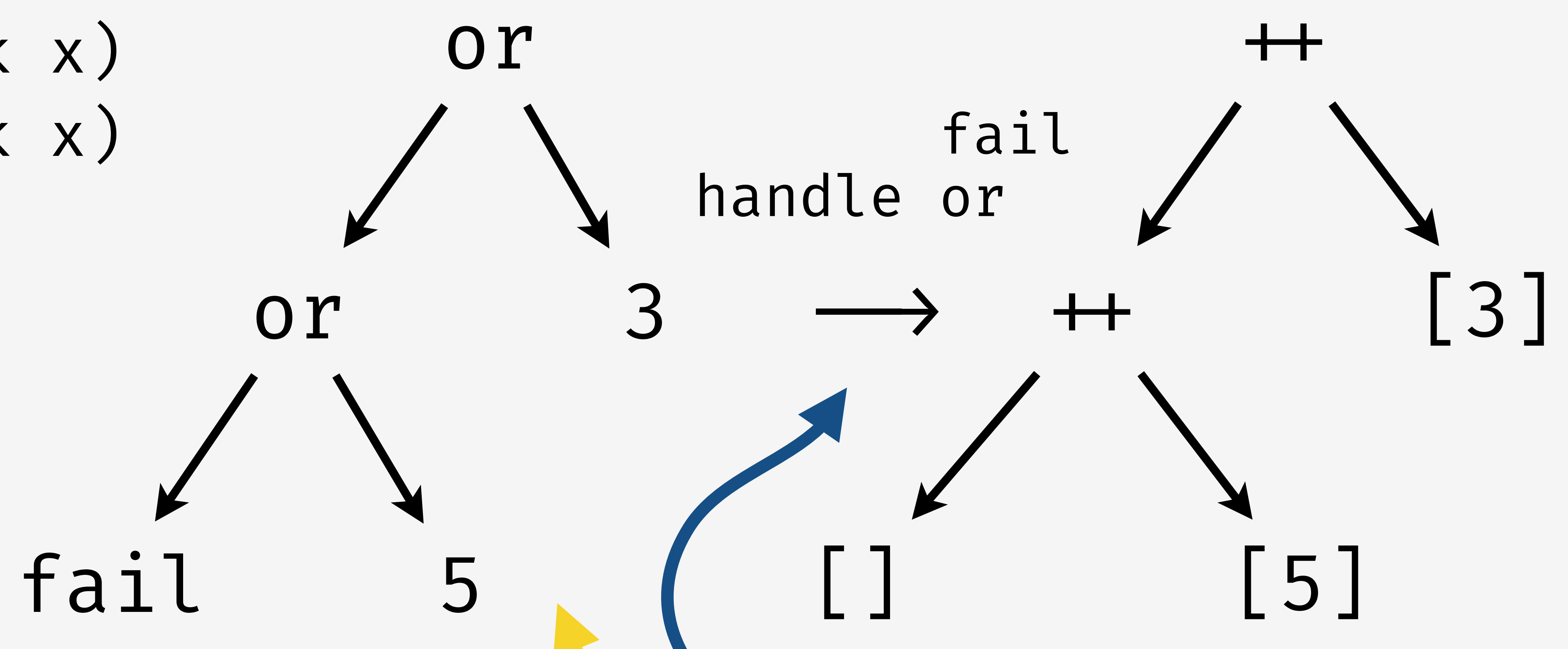
· **Algebraic:** Operations must respect substitution and sequencing

· **Syntactic** Programs are embedded syntax trees that can be inspected

· **Denotatative:** The semantics is given by a handler that replaces operations with their meaning

Algebraic Effect Handlers

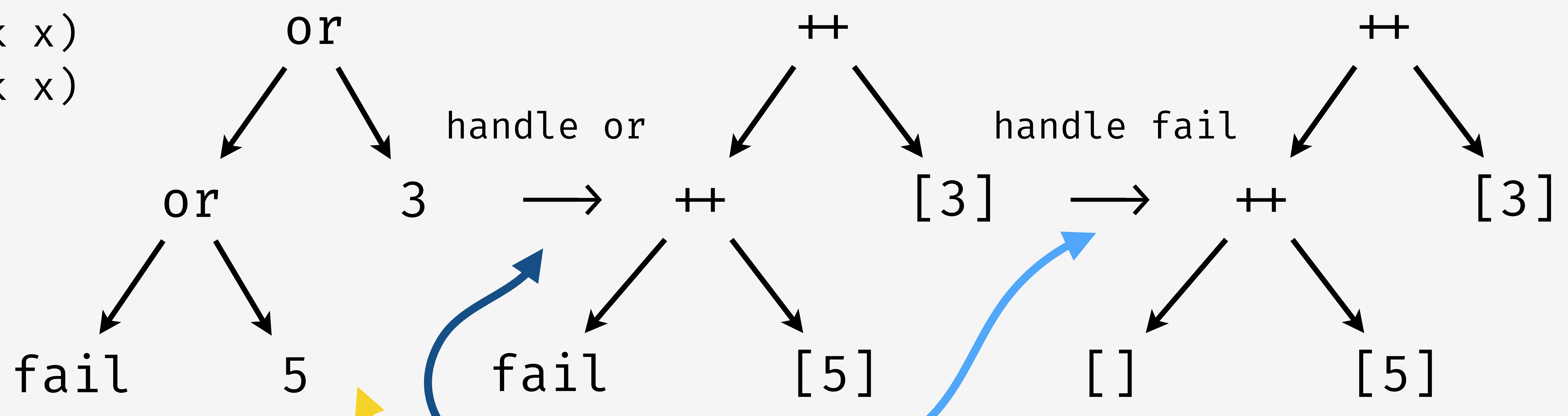
```
do x ← or p q     = do or (do x ← p; k x)
   k x                  (do x ← q; k x)

      do fail     =  do fail
         k ()
```



- **Algebraic:** Operations must respect substitution and sequencing
- **Syntactic** Programs are embedded syntax trees that can be inspected
- **Denotatative:** The semantics is given by a handler that replaces operations with their meaning
- **Extensible:** Syntax trees can be extended with new syntactic nodes

Algebraic Effect Handlers

```
do x ← or p q     = do or (do x ← p; k x)
   k x                    (do x ← q; k x)

     do fail  =  do fail
        k ()
```



· **Algebraic:** Operations must respect substitution and sequencing

· **Syntactic** Programs are embedded syntax trees that can be inspected

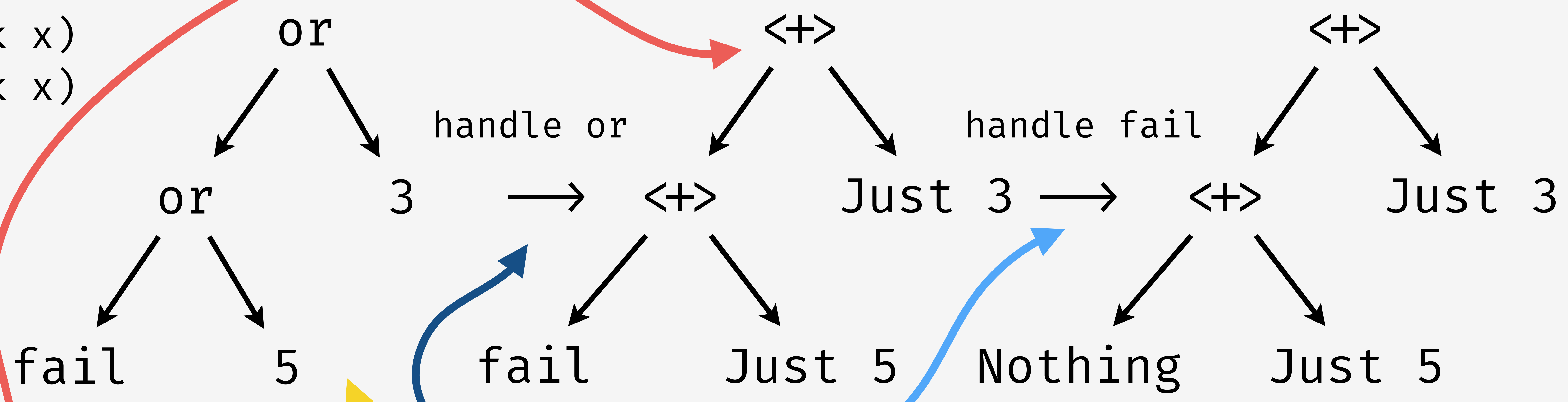
· **Extensible:** Syntax trees can be extended with new syntactic nodes

· **Denotatative:** The semantics is given by a handler that replaces operations with their meaning

· **Modular:** Handlers can be composed to give combinations of semantics

Algebraic Effect Handlers

```
do x ← or p q   = do or (do x ← p; k x)
   k x                  (do x ← q; k x)

      do fail  =  do fail
         k ()
```



· ***Algebraic:*** *Operations must respect substitution and sequencing*

· ***Syntactic*** *Programs are embedded syntax trees that can be inspected*

· ***Denotatative:*** *The semantics is given by a handler that replaces operations with their meaning*

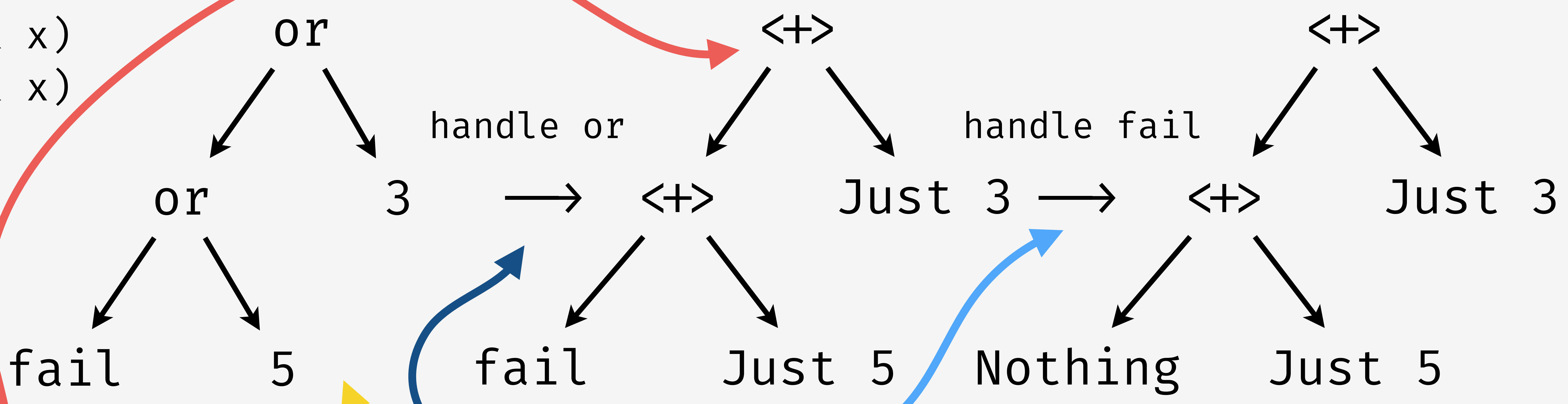
· ***Extensible:*** *Syntax trees can be extended with new syntactic nodes*

· ***Modular:*** *Handlers can be composed to give combinations of semantics*

· ***Flexible:*** *Multiple semantics can be given to a particular program*

Algebraic Effect Handlers

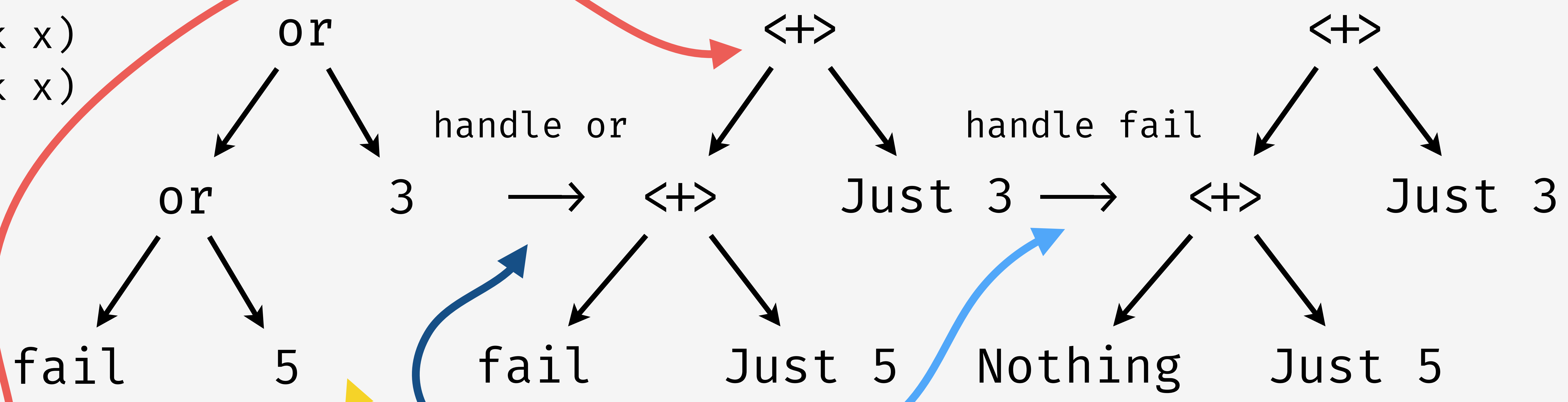
```
do x ← or p q    = do or (do x ← p; k x)
   k x                  (do x ← q; k x)

        do fail  =  do fail
           k ()

do x ← or p q  =  do x ← or q p
```



- ***Algebraic:*** *Operations must respect substitution and sequencing*
- ***Equational:*** *The relationship between operations can be expressed by laws*
- ***Syntactic*** *Programs are embedded syntax trees that can be inspected*
- ***Extensible:*** *Syntax trees can be extended with new syntactic nodes*
- ***Flexible:*** *Multiple semantics can be given to a particular program*
- ***Denotatative:*** *The semantics is given by a handler that replaces operations with their meaning*
- ***Modular:*** *Handlers can be composed to give combinations of semantics*

Algebraic Effect Handlers

```
do x ← or p q    = do or (do x ← p; k x)
   k x                   (do x ← q; k x)

      do fail    =  do fail
         k ()

do x ← or p q  =  do x ← or q p

print  get  put  throw  fork  read  write
```



· ***Algebraic:*** *Operations must respect substitution and sequencing*

· ***Equational:*** *The relationship between operations can be expressed by laws*

· ***Pervasive:*** *Algebraic effects cover a very large class of useful effects*

· ***Syntactic*** *Programs are embedded syntax trees that can be inspected*

· ***Extensible:*** *Syntax trees can be extended with new syntactic nodes*

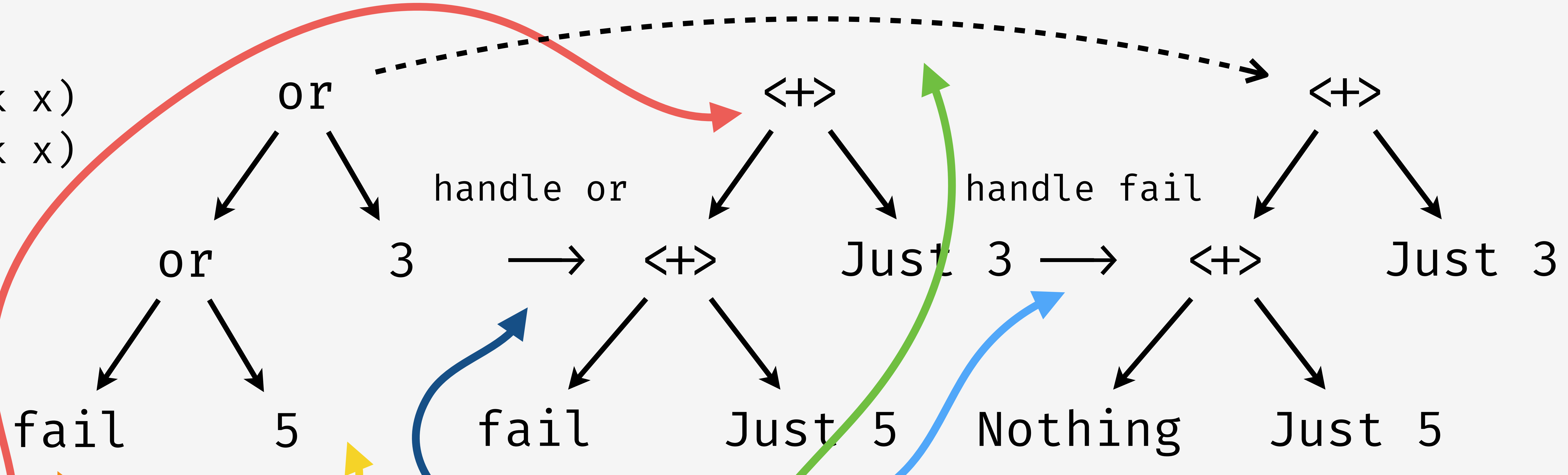
· ***Flexible:*** *Multiple semantics can be given to a particular program*

· ***Denotatative:*** *The semantics is given by a handler that replaces operations with their meaning*

· ***Modular:*** *Handlers can be composed to give combinations of semantics*

Algebraic Effect Handlers

```
do x ← or p q    = do or (do x ← p; k x)
   k x                   (do x ← q; k x)

      do fail    =  do fail
         k ()

do x ← or p q  =  do x ← or q p

print  get  put  throw  fork  read  write
```



- **Algebraic:** *Operations must respect substitution and sequencing*
- **Equational:** *The relationship between operations can be expressed by laws*
- **Pervasive:** *Algebraic effects cover a very large class of useful effects*
- **Syntactic** *Programs are embedded syntax trees that can be inspected*
- **Extensible:** *Syntax trees can be extended with new syntactic nodes*
- **Flexible:** *Multiple semantics can be given to a particular program*
- **Denotatative:** *The semantics is given by a handler that replaces operations with their meaning*
- **Modular:** *Handlers can be composed to give combinations of semantics*
- **Efficient:** *Intermediate trees can be avoided, to immediately return results*

Handlers and Heuristics

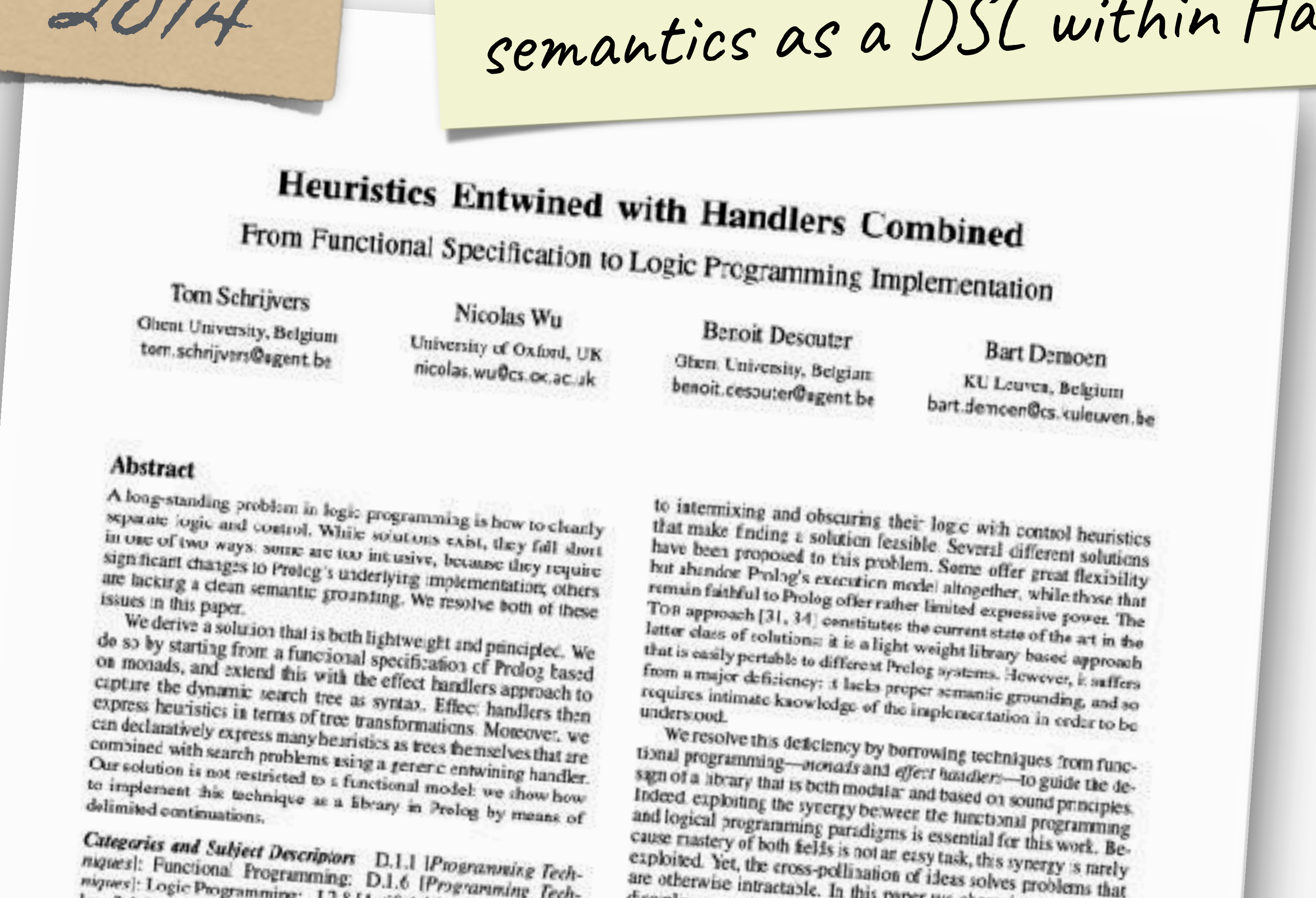
In a tree of nondeterministic computations, there are many different evaluation strategies

For instance, breadth-first, depth-first, depth-bounded, single result etc.

This paper shows how these different strategies are handlers of nondeterminism

This was used to model Prolog semantics as a DSL within Haskell

2014



Heuristics Entwined with Handlers Combined

From Functional Specification to Logic Programming Implementation

Tom Schrijvers
Ghent University, Belgium
tom.schrijvers@ugent.be

Nicolas Wu
University of Oxford, UK
nicolas.wu@cs.ox.ac.uk

Benoit Desouter
Ghent University, Belgium
benoit.desouter@ugent.be

Bart Demoen
KU Leuven, Belgium
bart.demoen@cs.kuleuven.be

Abstract

A long-standing problem in logic programming is how to clearly separate logic and control. While solutions exist, they fall short in one of two ways: some are too intrusive, because they require significant changes to Prolog's underlying implementation; others are lacking a clean semantic grounding. We resolve both of these issues in this paper.

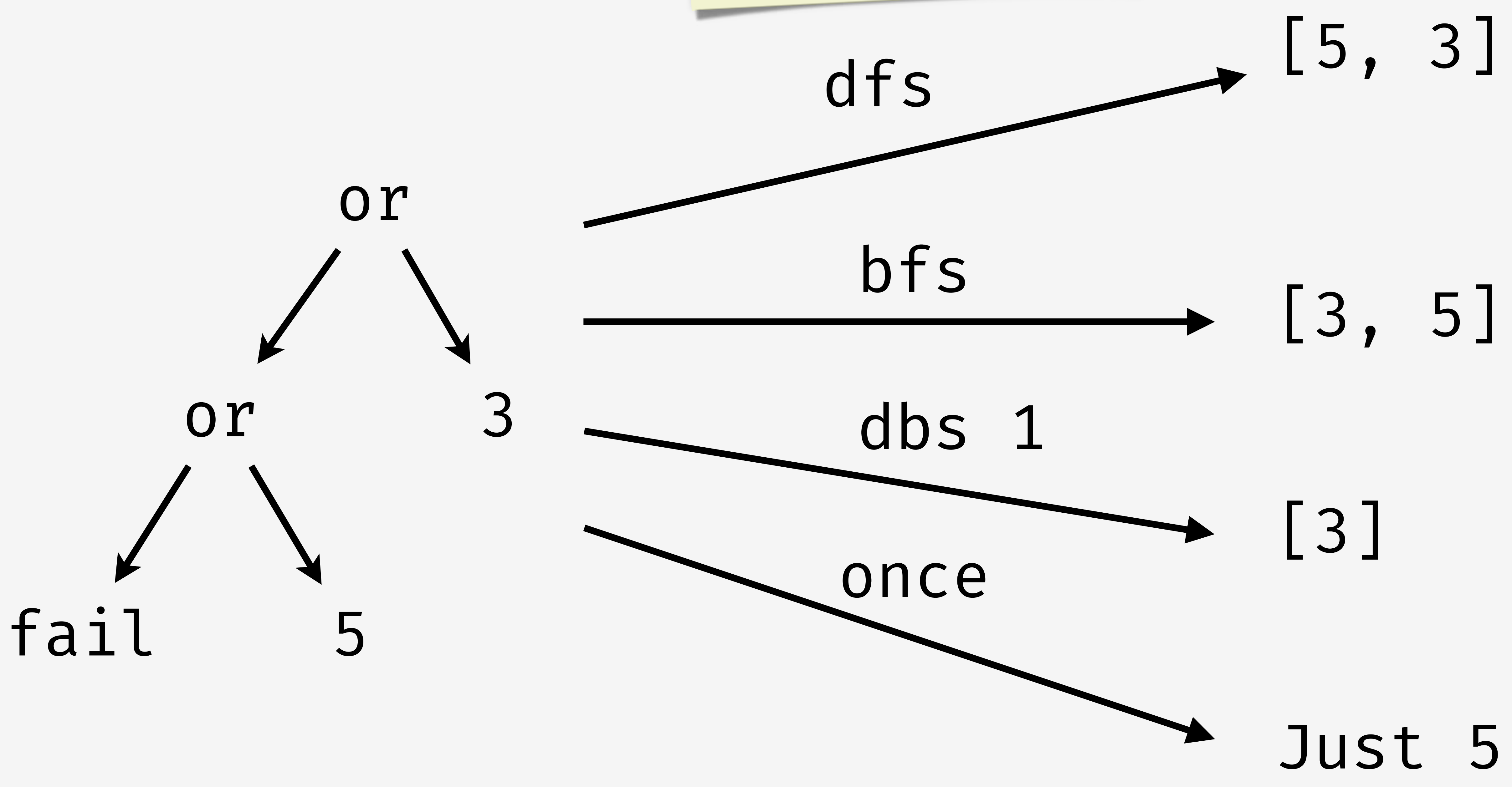
We derive a solution that is both lightweight and principled. We do so by starting from a functional specification of Prolog based on monads, and extend this with the effect handlers approach to capture the dynamic search tree as syntax. Effect handlers then express heuristics in terms of tree transformations. Moreover, we can declaratively express many heuristics as trees themselves that are combined with search problems using a generic entwining handler. Our solution is not restricted to a functional model: we show how to implement this technique as a library in Prolog by means of delimited continuations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; D.1.6 [*Programming Techniques*]: Logic Programming; ...

to intermixing and obscuring their logic with control heuristics that make finding a solution feasible. Several different solutions have been proposed to this problem. Some offer great flexibility but abandon Prolog's execution model altogether, while those that remain faithful to Prolog offer rather limited expressive power. The Tor approach [31, 34] constitutes the current state of the art in the latter class of solutions: it is a light weight library based approach that is easily portable to different Prolog systems. However, it suffers from a major deficiency: it lacks proper semantic grounding, and so requires intimate knowledge of the implementation in order to be understood.

We resolve this deficiency by borrowing techniques from functional programming—*monads* and *effect handlers*—to guide the design of a library that is both modular and based on sound principles. Indeed exploiting the synergy between the functional programming and logical programming paradigms is essential for this work. Because mastery of both fields is not an easy task, this synergy is rarely exploited. Yet, the cross-pollination of ideas solves problems that are otherwise intractable. In this paper ...

each semantics is given by a handler



Scoped Effects

Many handlers have related behaviour

```
list (return x) = [x]
list (fail)     = []
list (or p q)   = list p ++ list q
```

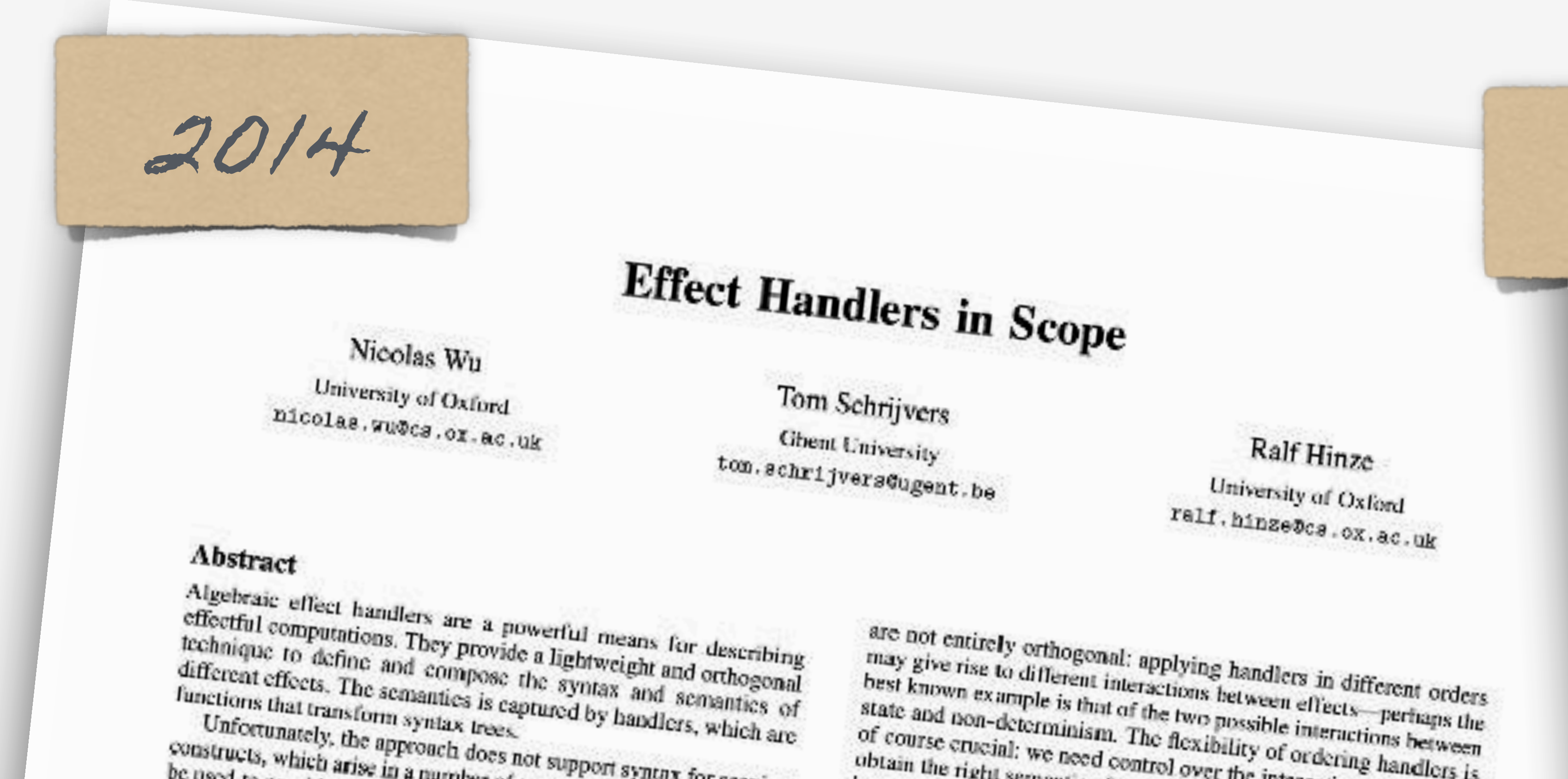
```
once (return x) = Just x
once (fail)     = Nothing
once (or p q)   = case once p of
                    Nothing → once q
                    Just x  → once p
```

These are not algebraic operations

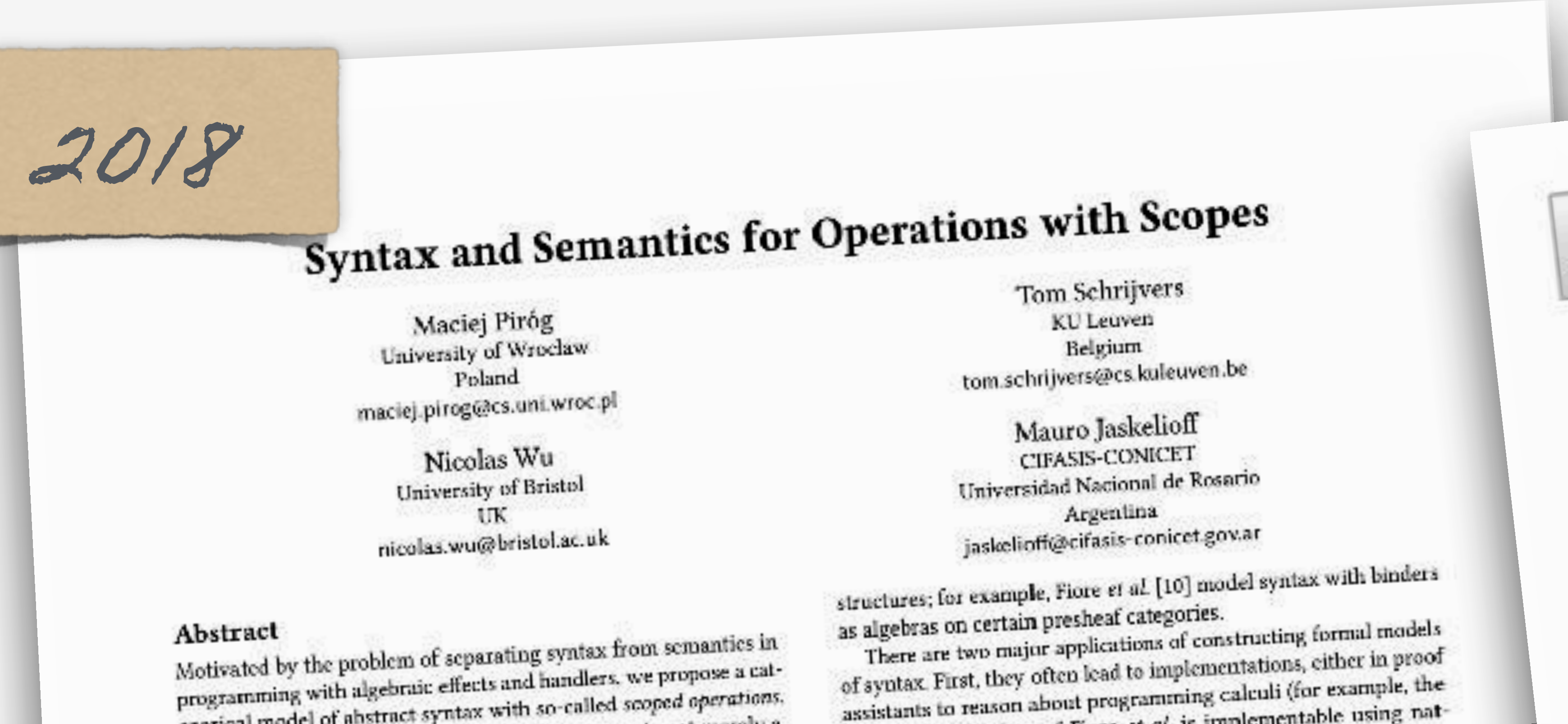
Can the idea of algebraic operations be extended?

```
search (fail)       = fail
search (return x)   = return x
search (search p)   = search p
```

2014



2018



2022

Theory Meets Practice

GitHub

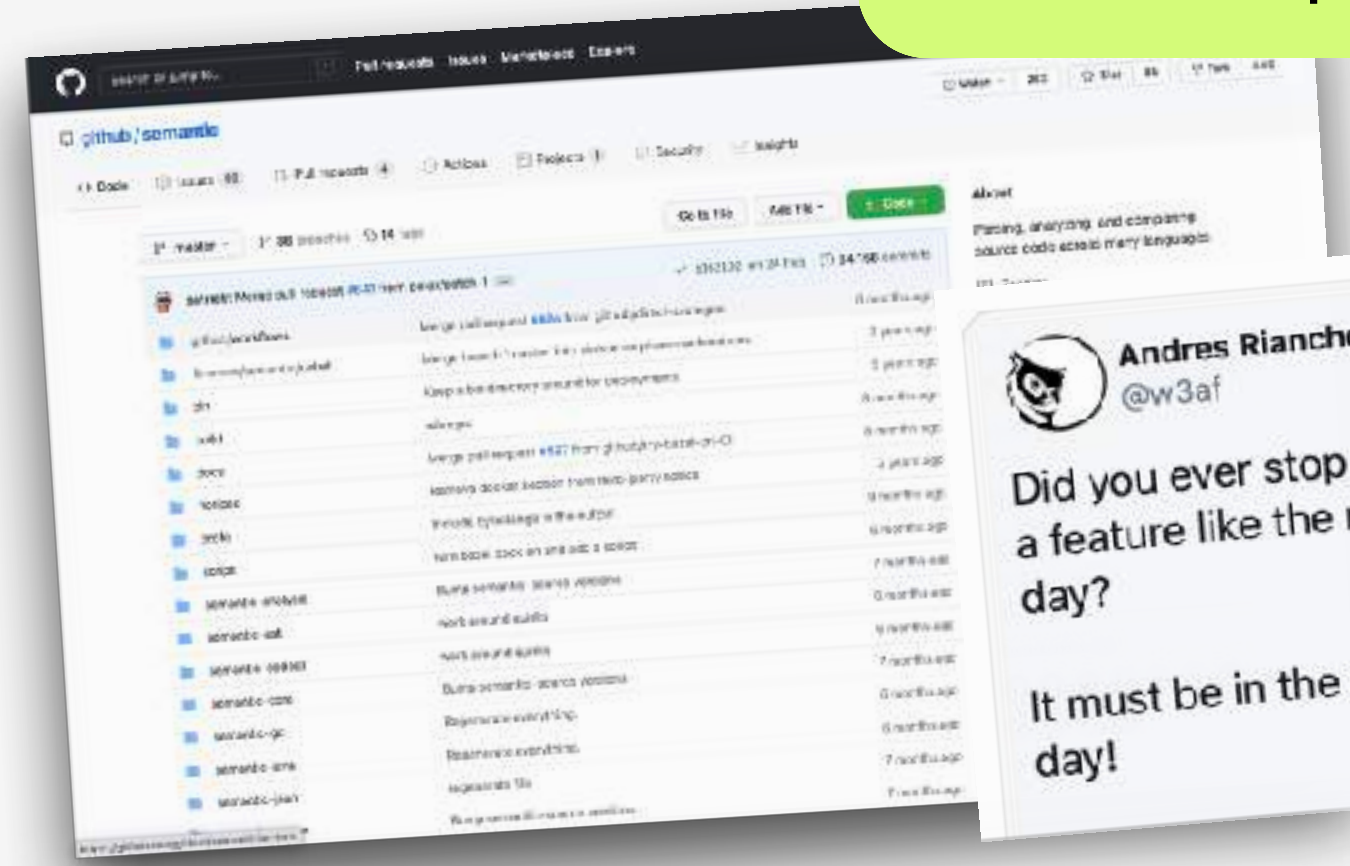
Algebraic effects are nice, but we can't express lots of constructs like if statements and try/catch as syntax.

Hmm, sounds like you need scoped effects ...

Wow! That works, but is it efficient?

Yes! It all fuses!

Amazing! 250x faster than our previous attempts! We've rolled this out to production!



functional

Alonzo Church

ISWIM brings into sharp relief some of the distinctions that the author thinks are intended by such adjectives as procedural, nonprocedural, algorithmic, heuristic, imperative, declarative, functional, descriptive. Here is a suggested classification, and one new word.

Effect handlers?

denotative!

Peter Landin

> imperative

Alan Turing

References

- M. Davis, Engines of Logic, 2000
- M. Davis, The Universal Computer: The Road from Leibniz to Turing, Third Edition, 2018
- A. Hodges, Alan Turing: The Enigma, 2014
- D. Turner, Some History of Functional Programming Languages, 2012
- G. Leibniz, De Arte Combinatoria, 1666
- D. Hilbert, W. Ackermann, Grundzüge der theoretischen Logik, 1928
- J. Backus et al., The FORTRAN Automatic Coding System for the IBM 704, 1956
- F. Cajori, A History of Mathematical Notations, 1928
- K. Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, 1931
- A. Church, A Set of Postulates for the Foundation of Logic, 1932
- A. Church, A Note on the Entscheidungsproblem, 1936
- A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, 1937
- J. von Neumann, First Draft of a Report on the EDVAC, 1945
- J. McCarthy et al., LISP I Programmer's Manual, 1960
- J. Backus et al., Revised Report on the Algorithmic Language Algol 60, 1963
- P. Landin, The mechanical evaluation of expressions, 1964
- P. Landin, A correspondence between ALGOL 60 and Church's Lambda-notation: Part I, 1965
- D. Scott, C. Strachey, Towards a Mathematical Semantics for Computer Languages, 1971
- P. Landin, The Next 700 Programming Languages, 1966
- R. Hinze, N. Wu, J. Gibbons, Unifying Structured Recursion Schemes, 2013
- R. Hinze, N. Wu, J. Gibbons, Conjugate Hylomorphisms, 2015
- Z. Yang, N. Wu, Fantastic Morphisms and Where to Find Them, 2022
- G. Plotkin, M. Pretnar, Hanlders of Algebraic Effects, 2009
- T. Schrijvers, N. Wu, B. Desouter, B. Demoen, Heuristics Entwined with Handlers Combined, 2014
- N. Wu, T. Schrijvers, R. Hinze, Effect Handlers in Scope, 2015
- M. Pirog, T. Schrijvers, N. Wu, M. Jaskelioff, Syntax and Semantics for Operations with Scopes, 2018
- N. Wu, T. Schrijvers, Fusion for Free, 2015
- P. Thomson, R. Rix, N. Wu, T. Schrijvers, Fusing Industry and Academia at GitHub, 2022

with thanks to Jeremy Gibbons and Jamie Willis
for feedback on previous iterations of this talk